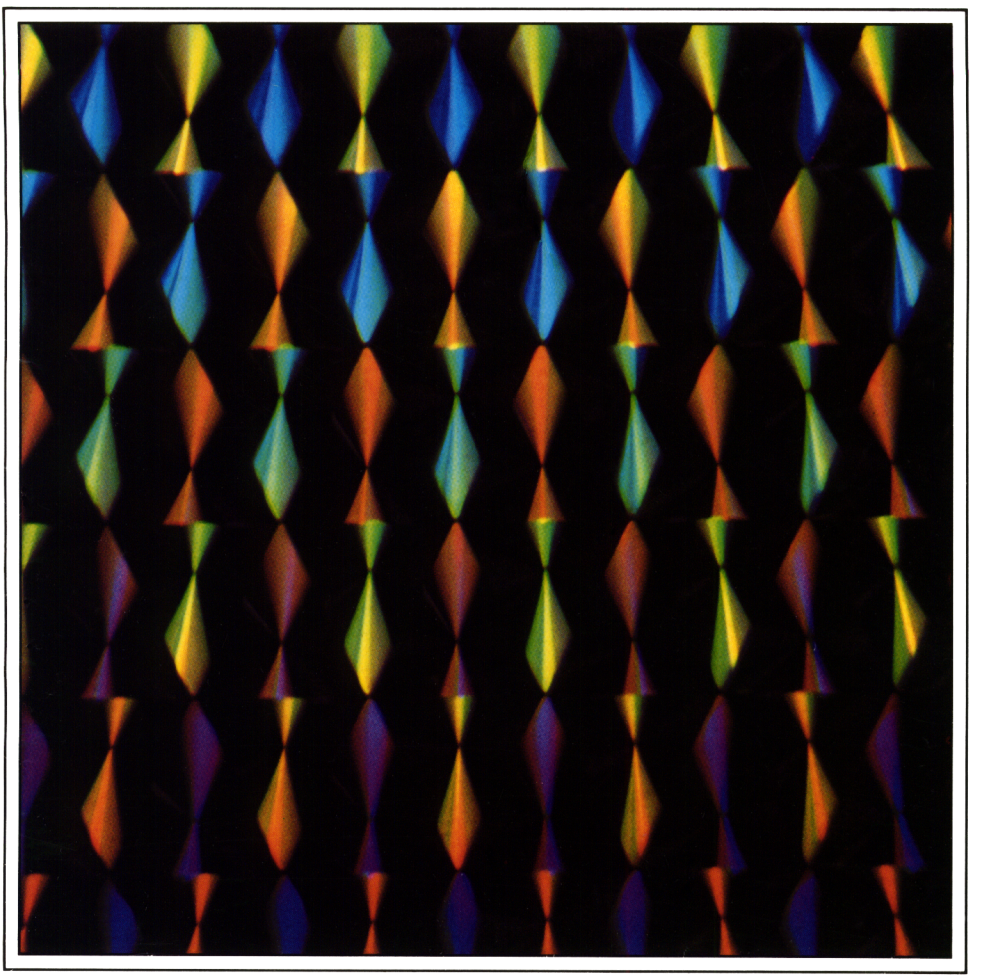
BBC
NEC

Structured Basic

A Guide to Good Programming Style for the
British Broadcasting Corporation Microcomputer



Richard Freeman

STRUCTURED BASIC

A Guide to Good Programmming Style
For the British Broadcasting Corporation Micro

Richard Freeman

British Broadcasting Corporation
National Extension College

Author's acknowledgements
It is a great pleasure to acknowledge the enormous help that Ian Trackman gave me in the preparation of this book. He painstakingly checked through the text and programs, spotting errors and ambiguities and he provided suggestions for additional material. Whatever its faults, the book is immensely better as a result of Ian Trackman's help.

Errors and faults remain. They are my responsibility and I would welcome letters from users of the book pointing out errors and suggesting improvements.

© Richard Freeman 1984
First published 1984

Published by the British Broadcasting Corporation
35 Marylebone High Street
London W1M 4AA

and

National Extension College
18 Brooklands Avenue
Cambridge CB2 2HN

ISBN 0 563 21002 8 (BBC)
ISBN 0 86082 397 0 (NEC)

Contents

		Page
Unit 1	Program style	8
Unit 2	Repetitions and decisions	30
Unit 3	Procedures and user-defined functions	69
Unit 4	Graphics	102
Unit 5	Program planning	140
Unit 6	Data vetting and data entry	163
Unit 7	Files	195
Unit 8	Data bases	231
Unit 9	Games	261
Unit 10	Testing and de-bugging	299
	Appendix 1 BBC Micro character set	312
	Appendix 2 Polar coordinates	313
	Appendix 3 Structured programming in minimal BASIC	315

INTRODUCTION

How to write better programs

Once you start to write moderately long programs in BASIC, you may have found that your programs tend to get out of control. This book is about bringing them back under control. How can we do that?

The key to keeping a program under control lies in:

- ☐ Good advance planning to ensure that the program can be built in small steps which are as independent of each other as possible.
- ☐ Anticipating future changes and developments.
- ☐ Using 'top-down' programming design methods.
- ☐ Structuring the program so that it is easy to control its flow.

All of these things can be done when writing a program in any computer language, but if a language has certain features, it is more suitable for writing structured programs. The most important features that we need are all available in BBC BASIC: `FOR ... NEXT`, `REPEAT ... UNTIL ...`, `IF ... THEN ... ELSE ...`, procedures and user-defined functions. This book shows you how to use these to produce clear, understandable, effective and flexible programs. But even if your version of BASIC doesn't have these features, you can still write structured programs. The special adjustments that you need to do this are described in Appendix 3. We shall tell you when to refer to it as you work through the book.

What is the problem?

We need to find methods for the design and writing of programs which will produce reliable results under all circumstances. Most books and popular computing magazines don't even mention that such methods are needed. Other books tell you that there is one perfect method or that you can only write good programs in one particular language. In this book, we try to recognise the problem but not to be too dogmatic about the solution. In particular, we have tried to start from where you are likely to be and to build onto what you already know and do. You will therefore find flowcharts in this book (some dogmatists say that they should never be used) and you will not find structure diagrams or 'Kludge' code (some dogmatists say that you can't write good programs without them). We shall not tell you what to think on these academic issues. When you have worked through this book you will be in a position to join the argument and to decide on your own stand. Meanwhile, your program design should have improved immensely through an evolution of your techniques, not a revolution. We shall ask you to take some revolutionary steps such as giving up `GOTO` (not just for Lent, but for ever!) but mostly we shall suggest gentle changes. That's all that most of us can take in life.

Now let's return to the problem and to the approach suggested by the title of the book. Unless a program is very short, what it is doing and how it is doing it will only be clear if the author has taken very careful steps to make it so. The program has a structure. Is it clear what that structure is? Is that structure sensible? An example should show what we mean. Suppose that a doctor requires a program to maintain patients' records. Guessing at what he would need to be able to do, the structure of the program would need to be something like:

```
                              MENU
Add new          Delete old          Update a         Search
patient's        patient's           patient's        records
details          details             record
```

When the program is eventually coded, that structure should be absolutely clear. How can we make it so? That takes us back to the general principles of good programming:

- ☐ Analyse the problem to be solved.
- ☐ Design a solution with a structure that clearly reflects the solution to the problem.
- ☐ Code the solution, preserving the structure so that it is clearly displayed in the coded program.
- ☐ Document the program clearly.
- ☐ Test the program systematically to ensure that it does solve the original problem.

Principles are not enough to generate good results. Some kind of method or guideline is also needed. Two such methods will be used throughout this book: top-down design and modular design.

Top-down design involves solving a problem in a series of steps, each of which is a refinement of the previous step. The patient's records design above has been solved in four sections. Each of these sections can now be refined into smaller steps and so on, until we are satisfied that no further detail is needed. Units 1 to 4 of this book lead up to your being able to apply top-down design to a long program and this is done in Unit 5.

Modular design requires that we carry out top-down design in a way which requires the use of self-contained sections of program: an input module, a histogram drawing module, a circle drawing module etc.

So we are aiming at programs with

- ☐ a clear structure
- ☐ a top-down design
- ☐ modular form

As a set of principles, that's fine, but BASIC is against us! BASIC allows us to write programs with appalling structures because (amongst other things) it allows us to abuse loops and to jump all over the place with

GOTO. So, if we are to keep to the above principles, we have to impose some rules of self-discipline. The one that needs emphasis here is the need to abolish GOTO. Whenever we switch program flow, the reason has to be clearly stated. If it is not, the program will not have a clear structure.

```
IF income > 50000 THEN 350
```

is meaningless.

```
IF income > 50000 THEN PROCtax_heavily
```

is all too clear! Clarity is what we want. GOTO and clarity do not go together.

To summarise, then, a good program must have a well designed structure and that requires careful advance planning. But once we have designed a good structure, we must code the program in a way that preserves and even amplifies that structure. That is what we hope to achieve in this book.

How to use this book

This book is designed to help you to teach yourself. Most of what you learn will come from what you do – and there is plenty to do in this book:

- ☐ First, try all the programs.
- ☐ Second, try varying the programs until you are sure that you know how they work.
- ☐ Third, work carefully through all the Self Assessment Questions (SAQs). These are designed to help you to learn in small steps. Answers to SAQs appear at the end of the unit in which the SAQ appears. Make sure that you understand each new point as it is introduced.

What you need to know to start

This book assumes that you already know how to use the commonest BASIC statements and functions:

```
ABS
CLS
DATA
DIM (for one- and two-dimensional arrays)
END
FOR ... NEXT
GOSUB
GOTO
IF ... THEN ...
INPUT
```

```
INT
LEFT$
LEN
LET
LIST
LOAD
MID$
PRINT
READ
REM
RETURN
RIGHT$
RND
RUN
SAVE
SQR
STEP
STOP
STR$
TAB
VAL
+, -, *, /, ^, =, <, >
```

Such a range of statements is found in most introductory BASIC texts including *30 Hour BASIC*, which this book is specifically designed to follow.

Assignments
If you are using this book as part of a National Extension College correspondence book or a local FlexiStudy book or in a college class, you will find that Units 1 to 9 include an assignment designed for tutor marking. Answers to the assignments are not included in the book.

BBC BASIC
All programs in this book are written in BBC BASIC for use on a British Broadcasting Corporation Model B Micro. Most will also run on a Model A. Part of Unit 7 and the whole of Unit 8 require the use of a disk filing system.

Apart from the problem of graphics, most programs can be readily altered to run on micros with other dialects of BASIC. Changing the graphics routines is a little harder but not too difficult once you have found the equivalent statements in your BASIC to BBC BASIC's `MOVE`, `DRAW`, `PLOT` etc.

The User Guide
Throughout this book we shall assume that you have access to *The BBC Microcomputer User Guide*. This will be referred to as *The User Guide*.

UNIT 1

Program style

1.1 Style and structure

1.2 Letting some space in

1.3 Constants, variables and more spaces

1.4 Rules - conclusion

1.5 The end of GOTO

1.6 Flags and TRUE and FALSE

1.7 Integer variables

1.8 Style and structure

Assignment A

Objectives of Unit 1

Answers to SAQs

1.1 Style and structure

BASIC produces results very quickly. This has made it a very popular beginner's programming language. It has also resulted in thousands of bad programs being written and published. What do we mean by 'bad'? Unless you are writing a short program to do an immediate job with the intention of immediately scrapping the program, your program should meet certain criteria:

- ☐ It should be easy for you to read.
- ☐ It should be easy for others to read.
- ☐ Its purpose should be clear.
- ☐ Its structure should be clear.
- ☐ It should be easy to amend to do a slightly different job.
- ☐ It should be easy to extend into a much larger program.

All these points concern style and structure. We shall develop these as the book proceeds. As we introduce new BASIC statements and structures, we shall use them to improve the style of the programs that we write.

In this unit, we look at what we can do to improve the style of programs within the limits of the BASIC that you will have met so far.

1.2 Letting some space in

If you take away the white space around the letters on a page, you would have great difficulty in reading anything. Space is the most important part of any displayed message – just look how advertisers use white space on hoardings. This also applies to programs. A program without space is a program designed not to be read. Have a look at Program 1.1.

```
 10 DIM N(20),F(20)
 20 FOR I=1 TO 20
 30 READ N(I)
 40 NEXT I
 50 S=0
 60 FOR I=1 TO 20
 70 S=S+N(I)
 80 NEXT I
 90 PRINT"MEAN=";S/20
100 FOR I=1 TO 20
110 F(N(I))=F(N(I))+1
120 NEXT I
130 M=0
140 FOR I=1 TO 20
150 IF F(I)>M THEN M=I
```

```
160 NEXT I
170 PRINT"MODE=";M
180 FOR I=1 TO 19
190 FOR J=I+1 TO 20
200 IF N(I)<=N(J) THEN 230
210 T=N(I):N(I)=N(J):N(J)=T
220 NEXT J
230 NEXT I
240 PRINT"MEDIAN=";(N(10)+N(11))/2
250 END
260 DATA 20,6,5,5,5,9,1,19,20,11,20,7,5,3,9,1,4,10,9,14
```

Program 1.1

It's a very simple program and only 26 lines long – yet it is very hard either to read it or to tell how it flows and what it does. It clearly needs an improved layout. The quickest way to improve the layout is to re-list it on the BBC Micro with `LISTO 7`. `LISTO` (see *User Guide* page 290) controls the format in which you see programs on the screen and in which they are printed out by the printer. It inserts spaces into the listing but not into the program as stored on cassette or disk. `LISTO` can be followed by the numbers 0 to 7, each number using a slightly different set of spacing rules. Program 1.1 is listed in `LISTO 0`. Program 1.2 shows the same program after typing:

```
>LISTO 7
>LIST
```

```
 10 DIM N(20),F(20)
 20 FOR I=1 TO 20
 30   READ N(I)
 40   NEXT I
 50 S=0
 60 FOR I=1 TO 20
 70   S=S+N(I)
 80   NEXT I
 90 PRINT"MEAN=";S/20
100 FOR I=1 TO 20
110   F(N(I))=F(N(I))+1
120   NEXT I
130 M=0
140 FOR I=1 TO 20
150   IF F(I)>M THEN M=I
160   NEXT I
170 PRINT"MODE=";M
180 FOR I=1 TO 19
190   FOR J=I+1 TO 20
200     IF N(I)<=N(J) THEN 230
```

```
210      T=N(I):N(I)=N(J):N(J)=T
220      NEXT J
230    NEXT I
240 PRINT"MEDIAN=";(N(10)+N(11))/2
250 END
260 DATA 20,6,5,5,5,9,1,19,20,11,20,7,5,3,9,1,4,10,9,14
```

Program 1.2

That looks a little better and, indeed, is the best we can do with the Micro's own layout system. Any further improvements must come from the way that we type the program in. We are now going to introduce some rules to improve layout and apply them to Program 1.2. Since we are going to be adding a lot of extra lines, we shall re-number the program as we change it. This is done by typing:

RENUMBER or REN.

RULE 1
Start the program with a title.

A title is useful at a later stage; for instance, if you want to refer to the program in documentation to go with it, or in a catalogue of what is on a given disk or cassette. The shorter the better is probably the case here. We shall call this program 'STATISTICS', so we add a new line at the start of the program:

```
REM STATISTICS.
```

RULE 2
Give the program a description.

After the title, a short description of what the program does will help the reader to understand the rest of the program – or to decide that the program is of no use to him. Since Program 1.2 calculates the mean, median and mode of a set of figures, we add as the next line

```
REM Calculates mean, mode and median
```

as the description. It might also be helpful to add REM lines to indicate

The author (and a copyright notice)
The source of any algorithms or other ideas used in the program
The version number and date

RULE 3

Insert a blank line between every logical block of code.

In writing and printing, paragraphs are important: one idea, one paragraph is the rule. So too, in program writing. Each separate block of code that does one coherent thing should be identified and a blank line inserted above and below it. On the BBC Micro a blank line is achieved by typing the line number followed by a space (i.e. a press of the space bar). Without the space, the line number is erased on listing, but with the space it stays in the program. (Some micros do not accept a line consisting of a single space, in which case, use a colon or even a **REM** to create spacing). For this program, we have identified:

- ☐ Dimensioning the arrays.
- ☐ Reading the data into the array.
- ☐ Calculating the total sum.
- ☐ Calculating and printing the mean.
- ☐ Calculating the frequency of each number.
- ☐ Finding the mode (the commonest value).
- ☐ Printing the mode.
- ☐ Sorting the data.
- ☐ Identifying and printing the median (the middle value).

RULE 4

Head each coherent block with a description of its action and describe the action of the program fully with REMs.

Here is Program 1.2 with Rules 1 to 4 applied to it and then re-numbered:

```
10 REM STATISTICS
20
30 REM Calculates mean, mode and median
40
50 DIM N(20),F(20)
```

```
 60
 70 REM Read data into array
 80 FOR I=1 TO 20
 90   READ N(I)
100   NEXT I
110
120 REM Calculate total of data
130 S=0
140 FOR I=1 TO 20
150   S=S+N(I)
160   NEXT I
170
180 REM Calculate and print mean
190 PRINT"MEAN=";S/20
200
210 REM Calculate frequency of each number
220 FOR I=1 TO 20
230   F(N(I))=F(N(I))+1
240   NEXT I
250
260 REM Find commonest value
270 M=0
280 FOR I=1 TO 20
290   IF F(I)>M THEN M=I
300   NEXT I
310
320 PRINT"MODE=";M
330
340 REM Sort data into ascending order
350 FOR I=1 TO 19
360   FOR J=I+1 TO 20
370     IF N(I)<=N(J) THEN 400
380     T=N(I):N(I)=N(J):N(J)=T
390     NEXT J
400   NEXT I
410
420 REM Identify and print median value
430 PRINT"MEDIAN=";(N(10)+N(11))/2
440
450 END
460 DATA 20,6,5,5,5,9,1,19,20,11,20,7,5,3,9,1,4,10,9,14
```

Program 1.3

What a transformation! Program 1.1 could be explained by pages of text. Program 1.3 explains itself. It tells you what is going to do, what it is doing and what it has done.

1.3 Constants, variables and more spaces

So far what we have done is to re-display a 27 line program (Program 1.1) as a 47 line program (Program 1.3). We've moved it from the cramped small ads to the front page. For a program of that length, we might consider the self-documentation that we've built into it to be sufficient. Experience shows, however, that programs don't get left alone. No sooner are they used than someone wants to amend or extend them. Such amendments require a knowledge of three things:

- ☐ What the constants in the program mean. (Constants are numbers or strings which stay fixed during the run).
- ☐ What variable names have been used. (So that we don't use them for something else).
- ☐ What the variables refer to.

Because this information is essential before you (or someone else) start to amend a program, it is good practice to describe the constants and variables near the top of the program.

RULE 5

Describe the constants and variables.

Here is how we would do it for Program 1.3.

```
 10 REM STATISTICS
 20
 30 REM Calculates mean, median and mode
 40
 50 REM Program constants
 60 REM 20= Number of data items
 70
 80 REM Program variables
 90 REM N( )=data items
100 REM F( )=frequency of data values
110 REM M=mode
120 REM T=temporary value in sort
130 REM I and J=loop counters
140 REM S=sum of data items
150
160 DIM N(20),F(20)
                    etc.
```

However, we can help ourselves a lot more by applying two further rules:

RULE 6

> Define constants at the start of the program. Then use variable names for the constants in the rest of the program.

To see the reason for this rule, consider what we have to do if we decide to use Program 1.3 for 30 numbers instead of 20. '20' which is a constant for this program appears in line 50 (twice), 80, 140, 190, 220, 280, 360 (as 19 i.e. 20 - 1), 370, 440 (as 10 and 11). So even in a short program, it is a major job to find all the occurrences of a constant and to change them correctly. To avoid this problem, we should define the constants at the top of the program e.g.

```
N = 20: REM Number of data items
```

and use 'N' for the constant in the rest of the program. Then we can change the program to cope with 30 items simply by changing the line which defines N to

```
N = 30
```

But we shall not use N because of the second of the two rules about constants and variables:

RULE 7

> Use meaningful variable names.

Variable names

Most BASICs limit variable names to one or two characters:

```
A3   Z5   RT
```

In BBC BASIC, variable names must follow these rules:

- ☐ They must start with a letter (upper or lower case) or _ or £.
- ☐ They may be up to 255 characters long.
- ☐ The second character onwards may be upper case, lower case, digit, _ or £.
- ☐ They must not contain punctuation (except _).
- ☐ They may not contain a space.
- ☐ They may not begin with a BASIC keyword in upper case although they may contain BASIC keywords, even in capital letters, after the first letter.

You may be used to writing something like

```
H = 57
```

where H represents the height of a wall. In BBC BASIC, you could as

well call your variable HEIGHT and say

```
HEIGHT = 57
```

Or, you might prefer lower case and say

```
height = 57
```

Whichever you choose, BASIC treats the upper-case variable HEIGHT as being totally different from the lower-case variable height. Although you could use both versions for two different purposes within the same program, it is not advisable since it is likely to cause confusion.

Suppose now that you wanted to give a new name to a variable L in a statement like

```
L = 63
```

where L = length of the wall. You might write

```
LENGTH = 63
```

However, this variable name starts with a BASIC keyword, LEN. The computer would therefore read your intended variable LENGTH as LEN followed by an invalid construction. One easy way round this is to stick to lower case:

```
length = 63.
```

Using only lower-case letters for variable names has the further advantage of clearly distinguishing them from the BASIC keywords. All variables in this book will be either single upper-case letters or longer names in lower case.

We have illustrated these points with numeric variables, but the same points apply to string variable names.

SAQ 1

Which of the following are valid variable names in BBC BASIC?

(a) A8
(b) AGE
(c) HEIGHT1
(d) HEIGHT 2
(e) LETTER$
(f) letter$
(g) 1YEAR

Applying Rule 7 to Program 1.3, let's re-name all our variables:

Old name	New name
N( )	data( )
F( )	frequency( )
M	mode
T	temp
S	sum
20 (the constant)	items

We have not renamed the loop counters. Sometimes it is helpful to do so, but where a loop counter is used within a loop only, a longer variable name adds little to clarity.

We can now replace all occurrences of our constant 20 with 'items' and all occurrences of the old single-letter variables with our new long variable names to produce Program 1.4.

```
 10 REM STATISTICS
 20
 30 REM Calculates mean, mode and median
 40
 50 REM Program constants
 60 items = 20 : REM Number of data items
 70
 80 REM Program variables
 90 REM data( ) = data items
100 REM frequency( ) = frequency of data values
110 REM mode = mode
120 REM T = temporary value in sort
130 REM sum = total of data items
140 REM I and J = counters in loops
150
160 DIM data(items), frequency(items)
170
180 REM Read data into data( )
190 FOR I = 1 TO items
200   READ data(I)
210   NEXT I
220
230 REM Calculate total of data
240 sum = 0
250 FOR I = 1 TO items
260   sum = sum + data(I)
270   NEXT I
280
290 REM Calculate and print mean
300 PRINT "MEAN = " ; sum / items
310
320 REM Calculate frequency of each number
330 FOR I = 1 TO items
340   frequency(data(I)) = frequency(data(I)) + 1
350   NEXT I
360
370 REM Find commonest value
380 mode = 0
390 FOR I = 1 TO items
```

```
400   IF frequency(I) > mode THEN mode = I
410   NEXT I
420
430 PRINT "MODE = " ; mode
440
450 REM Sort data into ascending order
460 FOR I = 1 TO items - 1
470   FOR J = I + 1 TO items
480     IF data(I) <= data(J) THEN 520
490     temp = data(I)
500     data(I) = data(J)
510     data(J) = temp
520     NEXT J
530   NEXT I
540
550 REM Identify and print median value
560 IF items MOD 2 = 0 THEN 580
570 PRINT "MEDIAN = " ; data(items / 2) : GOTO 590
580 PRINT "MEDIAN = " ;(data(items / 2) + data(items / 2 + 1))
    / 2
590
600 END
610 DATA 20, 6, 5, 5, 5, 9, 1, 19, 20, 11, 20, 7, 5, 3, 9, 1,
    4, 10, 9, 14
```

Program 1.4

If you look closely at Program 1.4, you will find two other changes. First, it now has spaces around '=', '+', '/', ';' and ':'. We have done this to improve readability. It produces a big enough improvement to justify making it a rule:

RULE 8

> Insert spaces around all keywords and variables, around arithmetic operators and around '=', ';' and ':'.

The second change has occured in the string swap (line 390 in Program 1.3) which now occurs on 3 lines in Program 1.4 (lines 500 - 520). Throughout our final listing in Program 1.4, we have stuck pretty firmly to one statement per line. Generally, we shall stick to Rule 9:

RULE 9

Only one BASIC statement per line.

However, let's immediately note one exception. Where a **REM** refers to one line only, that **REM** is best placed on the same line as the statement it refers to e.g.

```
items = 20 : REM Number of data items
```

RULE 10

Use brackets to clarify compound expressions.

We haven't had an example where this rule is of value, but the more complex programs become, the more it helps to clarify what is happening e.g.

```
IF (A = 10 AND B = 6) OR (Y = 0 AND Z = - 1) THEN ...
```

is easier to follow than

```
IF A = 10 AND B = 6 OR Y = 0 AND Z = - 1 THEN ...
```

RULE 11

Place all **DATA** statements at the end of the program.

This makes it easy to find the data and easy to amend it if required.

1.4 Rules – conclusion

You may be worried on two scores about the rules:

1. Must I always follow the rules?

No, no-one can lay down rules which are best for all programs for all time. But before you break them, use them. Once you know these rules well and can apply them intuitively, you will be in a position to make sensible judgements about when to break them. (Incidentally, we shan't apply every rule to every program in this book; e.g. in many programs we shan't list the variables and, on occasions, we shall use more than one statement per line).

2. Don't they use up a lot of memory?

As they stand, yes. REMs and spaces use up memory. But you can purchase a special program called a 'cruncher' which will remove spaces and REMs from your programs before you run them. This allows you to keep and work on the long, well-spaced, readable version but to run the condensed, incomprehensible version. For once, you really can have the best of both worlds. (A 'cruncher' is one of the many utility programs on the *Toolbox* software pack published by BBC Publications.)

1.5 The end of GOTO

Let's dispose of one other point before we leave this unit. You may have learnt to use GOTO as an essential part of BASIC. You may therefore be surprised to know that this book will teach you how to banish GOTO from your programs (except for its specialised use with ON ERROR GOTO). Why, you may ask, should you learn how to get rid of GOTO?

GOTO is one of the mechanisms for controlling the flow of a program. It is concerned with the logic of the program. Logic is of dubious value if it is not clear. A glance at many computing magazines will show you that most BASIC programs are not clear. They are obscure entanglements of GOTOs and are frequently referred to as 'spaghetti programs'. The more you use GOTO, the less clear your program can be.

We have used GOTO in Programs 1.1 to 1.4 but from Unit 2 onwards we shall try to eliminate GOTO. We hope that you will see that writing structured BASIC programs is, amongst other things, a challenge in exterminating GOTO. As long as a program has GOTO in it (except in ON ERROR GOTO), it remains in need of improvement.

For example, in Programs 1.1 to 1.4, a sequence with GOTO appeared. In Program 1.4 it appeared as

```
460 FOR I = 1 TO items - 1
470   FOR J = I + 1 TO items
480     IF data(I) <= data(J) THEN 520
490     temp = data(I)
500     data(I) = data(J)
510     data(J) = temp
520     NEXT J
530   NEXT I
```

If we are prepared to use more than one statement on a line, then we can eliminate the GOTO:

```
460 FOR I = 1 TO items - 1
470   FOR J = I + 1 TO items
480     IF data(I) > data(J)  THEN temp  = data(I)  :   data(I)
    =  data(J) : data(J) = temp
520     NEXT J
530   NEXT I
```

Is there a reward? Yes. Once all GOTOs are gone, there can be no more mis-directions caused by mistakes in line numbers. The program will genuinely follow the logic of your original algorithm. That may be wrong as well, but that's another story (which we leave to Unit 5).

SAQ 2

Program 1.5 is a typical minimal BASIC program. Re-write it using the rules in this unit.

```
10 INPUT"Amount of money to be invested "A
20 INPUT"Interest rate (%) "R
30 INPUT"Number of years "Y
40 PRINT"VALUE OF INVESTMENT AT END OF EACH YEAR"
50 PRINT"YEAR" TAB(19)"VALUE"
60 FOR I=1 TO Y
70 V=A*(1+R/100)↑I
80 PRINT;I,V
90 NEXT I
```

Program 1.5

1.6 Flags and TRUE and FALSE

TRUE and FALSE

BBC BASIC includes the two pseudo-variables, TRUE and FALSE. They are there for use in the decision making structures of BASIC. Each has a fixed numerical value assigned to it which you can verify by typing

```
PRINT TRUE
```

(which produces the result -1) and

```
PRINT FALSE
```

(which produces the result 0).

Whenever you use IF ... THEN ... you are implicitly using TRUE and FALSE: if the condition is true, the action after THEN is executed:

```
IF (condition) THEN (action)
```

Action only executed if condition is true.

Here are some examples of the uses of IF ... THEN ... that you are likely to have seen:

```
IF  A = 3 THEN 350
IF  N >= M THEN 900
IF  S = 0 THEN PRINT "No data to process."
```

The conditions are 'A = 3', 'N >= M' and 'S = 0'.

Conditions of this type are fine, but with the aid of TRUE and FALSE we can write programs with even clearer conditions. We know that any expression with the value -1 will be regarded as TRUE by the BBC Micro (some other micros use TRUE = 1) and that any expression with the value 0 will be regarded as FALSE. This leads to the idea of a 'flag'. Flags are best explained by an example:

```
10 REM Flag demonstration
20
30 INPUT "Enter your height in inches: " height
40
50 IF height > 72 THEN tall = TRUE
60 IF height <= 72 THEN tall = FALSE
70
80 IF tall = TRUE THEN PRINT "Pick on someone your own size."
90 IF tall = FALSE THEN PRINT "I'll protect you."
```

Program 1.6a

Comments

Line 50 If 'height' is greater than 72 then the variable 'tall' is assigned the value -1 (since TRUE = -1).

Line 60 If 'height' is less than or equal to 72 then the variable 'tall' is assigned the value 0 (since FALSE = 0).

So at line 70, 'tall' must either be -1 or 0: no other value is permitted by the program. It follows that 'tall' either has the value TRUE or the value FALSE. We can therefore use it in tests:

Line 80 The message is printed only if 'tall' is TRUE.

Line 90 The message is printed only if 'tall' is FALSE.

Now that's quite a useful advance on the decision making techniques that you have met before, but we can simplify the method even further. In the test in line 80

```
IF tall = TRUE THEN PRINT "Pick on someone your own size."
```

the '= TRUE' is not needed at all. You can simply write

```
IF tall THEN PRINT "Pick on someone your own size."
```

This is because any expression outside the range -1 to 1 after 'IF' is evaluated as TRUE. So, if A = 10, the condition 'IF A THEN ...' would be evaluated as TRUE. If you use flags (which we definitely advise), make sure that they only take the values -1 and 0. If you do not restrict your flags in this way, you may well find that you make mistakes in statements which use them.

What about simplifying line 90? Let's go back to TRUE and FALSE. Try typing

```
PRINT NOT TRUE
```

and

```
PRINT NOT FALSE
```

You should find that 'NOT TRUE' equals 0 (i.e. FALSE) and that 'NOT FALSE' equals -1 (i.e. TRUE). We can simplify line 90 to read

```
IF NOT tall THEN PRINT "I'll protect you".
```

So, the simplified Program 1.6a now reads:

```
10 REM Flag demonstration
20
30 INPUT "Enter your height in inches: " height
40
50 IF height > 72 THEN tall = TRUE
60 IF height <= 72 THEN tall = FALSE
70
80 IF tall THEN PRINT "Pick on someone your own size."
90 IF NOT tall THEN PRINT "I'll protect you."
```

Program 1.6b

Warning: the BBC BASIC 'NOT' is not a proper logical inverter. It is a *mathematical* NOT. So NOT 55 is not zero as it would be with a logical inverter. (In fact NOT 55 = -56 which is one added to the complement of 55). Once again, this demonstrates that, with BBC BASIC, it is best to keep to flags with values -1 and 0.

'tall' is called a 'flag'. A flag is a variable that takes the values 0 and -1 only. (As the programmer, you must ensure that it takes these values only). Provided that you ensure that your flags are limited to these values, you can use them directly in IF ... THEN ... and so produce immediately understandable tests. For example, you could have tests reading:

```
IF overdrawn  THEN ...
IF NOT fast   THEN ...
IF happy      THEN ...
```

In a program the size of Program 1.6b, the value of flags is hardly brought out. After all, we could have written the program more quickly without using flags. But as your programs get bigger, decisions will be taken further and further away from the event that first gave rise to the data for the decision, and so the program becomes more and more obscure. Flags remove this obscurity, allowing you to make tests of more English-like form whilst still knowing that the test will be carried out correctly.

SAQ 3

Re-write Program 1.7, putting in flags where appropriate.

```
 10 REM In the stars
 20
 30 CLS
 40 PRINT "THE BBC MICRO HOROSCOPE" '
 50 X = RND(2) - 2
 60 IF X = -1 THEN PRINT "Today is your lucky day."
 70 IF X = 0 THEN PRINT  "Today is not your lucky day."
 80
 90 X = RND(2) - 2
100 IF X = -1 THEN PRINT "You will meet an interesting
    stranger today."
110 IF X = 0 THEN PRINT "Be cautious about the strangers
    you meet today."
120
130 X = RND(2) - 2
140 IF X = -1 THEN PRINT "Today is a good day for a
    new business venture."
150 IF X = 0 THEN PRINT "You should avoid new business
    ventures today."
```

Program 1.7

The flags which we have created are being used as synthesised 'Boolean variables'. That is, variables that can only take the value 0 or the value -1. Some programming languages include genuine Boolean variables but in BBC BASIC the programmer has to ensure that his synthesised Boolean variables are restricted to the values 0 and -1.

1.7 Integer variables

In *30 Hour BASIC*, we introduced string and numeric variables. A typical numeric variable would be assigned a value as follows:

```
N = 42
```

The computer doesn't know that `N` is going to be a whole number so it stores it as a decimal and uses five memory locations (5 bytes) to do so –

more than is strictly necessary. If we know that a variable is going to be a whole number we can tell the computer that this will be so by placing % after the variable name:

```
N% = 42
```

or

```
SIZE% = 5
```

This has two advantages:

- ☐ % variables take up less memory space than ordinary numeric variables (although the % symbol itself requires an extra byte in your program every time that you use that variable in the code).
- ☐ BASIC can perform calculations faster with % variables than with floating point numeric variables (the type you may have previously used).

You can also have integer numeric arrays:

```
A%( )
```

which will only hold integer variables. Again, the main point of using the array A%( ) instead of A( ) is to speed up computing time and to save memory space.

```
10 DIM A(10)
20 FOR I = 1 TO 10
30   INPUT A(I)
40   NEXT I
```

Program 1.8a

will allow you to put ten decimal numbers into the array A( ). If you knew that all your numbers were to be integers, you might use A%( ) to save memory space:

```
10 DIM A%(10)
20 FOR I = 1 TO 10
30   INPUT A%(I)
40   NEXT I
```

Program 1.8b

The run time difference between Programs 1.8a and 1.8b is not determined by the use of % since the speed of execution will depend on the speed with which the user enters his data. Where speed is critical, you can use TIME to check how long a program takes – we show you how to do this in Unit 3. Meanwhile, if you know how to use TIME, you

may wish to compare the execution time of Programs 1.8c and 1.8d.

```
10 FOR I = 1 TO 5000
20   NEXT I
```

Program 1.8c

```
10 FOR I% = 1 TO 5000
20   NEXT I%
```

Program 1.8d

(Our run took 3.25 seconds for Program 1.8c and 1.11 seconds for Program 1.8d.)

SAQ 4

Rewrite Program 1.9 using % variables where appropriate:

```
 10 REM Build a daisy chain
 20 REM daises = No. of daisies
 30 REM length = length of chain
 40
 50 CLS
 60 length = 0
 70 FOR daisies = 1 TO 50
 80   length = length + 2.5 * RND(1)
 90   NEXT daisies
100 PRINT "Length of final chain = " ; length
```

Program 1.9

```
>RUN
Length of final chain = 56.3849735
```

1.8 Style and structure

In sections 1.1 to 1.4 we looked at improving the style of a simple program in order to make it more readable, easier to understand and easier to amend. However, the rules that we used would do little to improve a thoroughly bad program. If the initial structure of a program is poor, style rules will not be able to improve it. So, how can we produce good program structures? That is what the rest of this book is about, but some general points can be made at this stage.

Any program consists of three types of sequence:

- ☐ Serial sequences — e.g. line 300 of Program 1.4
- ☐ Branching instructions — e.g. line 480 of Program 1.4
- ☐ Loop sequences — e.g. lines 250 to 270 of Program 1.4.

We can use these to make the structure of a program clear or we can use them to create an awful mess. Designing a program in a way which makes the structure clear is what this book is about. In Units 2 and 3 we introduce a number of aspects of BBC BASIC that make it easier to write

well-structured programs. In Unit 4 we apply some of these to simple graphics routines and then in Unit 5 we bring all the ideas together and look at the design of programs. By the end of Unit 5 you will have learnt a wide range of techniques which will help in structuring long programs. In Units 6 to 9 we look at a variety of applications of these ideas.

Assignment A

1. Take any program which you have written (up to 30 lines). Re-write it using rules 1 - 9. Use integer variables and flags where appropriate.

Objectives of Unit 1

After studying this unit, you should be able to:

- ☐ Set out programs in a readable format by using:
  - LISTO 7
  - Blank lines
  - Titles
  - `REM`s with comments
  - Variable reference tables
  - Constant reference tables
  - Meaningful variable names
- ☐ Use flags in association with `TRUE`, `FALSE` and `NOT` to control program flow.
- ☐ Choose meaningful flag names.

Answers to SAQs

SAQ 1

The valid variable names are: (a), (b), (c) and (f).
The invalid ones are:
(d) which contains a space;
(e) which starts with the BASIC keyword '`LET`';
(g) which starts with a number.

SAQ 2

```
10 REM INVESTMENT
20
30 REM Calculates investment value after each year
40 REM for an investment with re-invested interest
```

```
 50
 60 REM Program variables
 70
 80 REM amount = initial value of investment
 90 REM rate = interest rate (%)
100 REM years = total period of investment
110 REM value = value of investment at end of each year
120 REM year = loop counter for years of investment
130
140 REM Enter data for investment
150
160 INPUT "Amount of money to be invested " amount
170 INPUT "Interest rate (%) " rate
180 INPUT "Number of years " years
190
200 REM Print headings
210
220 PRINT "VALUE OF INVESTMENT AT END OF EACH YEAR"
230 PRINT "YEAR" TAB(19) "VALUE"
240
250 REM Calculate and print investment values
260
270 FOR year = 1 TO years
280   value = amount * (1 + rate / 100) ↑ year
290   PRINT ; year, value
300   NEXT year
```

Program 1.10

SAQ 3

```
 10 REM In the stars
 20
 30 CLS
 40 PRINT "THE BBC MICRO HOROSCOPE" '
 50 lucky = RND(2) - 2
 60 IF lucky THEN PRINT "Today is your lucky day."
 70 IF NOT lucky THEN PRINT  "Today is not your
    lucky day."
 80
 90 stranger = RND(2) - 2
100 IF stranger THEN PRINT "You will meet an
    interesting stranger today."
110 IF NOT stranger THEN  PRINT "Be  cautious about
    the strangers  you meet today."
120
130 business = RND(2) - 2
```

```
140 IF business THEN PRINT "Today is a good day for
    a new business venture."
150 IF  NOT business  THEN PRINT  "You should  avoid
    new business ventures today."
```

Program 1.11

SAQ 4

```
 10 REM Build a daisy chain
 20 REM daisies% = No. of daisies
 30 REM length = length of chain
 40
 50 CLS
 60 length = 0
 70 FOR daisies% = 1 TO 50
 80   length = length + 2.5 * RND(1)
 90   NEXT daisies%
100 PRINT "Length of final chain = " ; length
```

Program 1.12

This was an exercise in detecting which variables can be integers and which cannot. ('length' cannot be an integer here since length is a continuous variable – i.e. it can take any decimal value). But variables followed by '%' are harder to read than those without, so it is probably best to avoid integer variables except when they are used for speed or to save memory space.

UNIT 2

Repetitions and decisions

2.1 REPEAT ... UNTIL ... and WHILE

2.2 IF ... THEN ... revisited

2.3 AND, OR, NOT and EOR

2.4 ON ... GOTO ...

2.5 Error trapping and reporting

Assignment B

Objectives of Unit 2

Answers to SAQs

Preliminary note
In Unit 1, we introduced integer variables using %. To give you more practice in using integer variables, we have used them extensively in some programs in this unit even when we don't need them either for space or speed. After this unit, we shall return to using integer variables more selectively.

2.1 REPEAT ... UNTIL ... and WHILE

The 'REPEAT ... UNTIL ...' loop will enable you to tidy up many programs which you have already met and to go on to develop complex programs which are nevertheless very easy to follow.

REPEAT ... UNTIL ... is very like FOR ... NEXT with some useful differences. It consists of two parts which will generally be several lines apart:

```
REPEAT

...

Action

...

UNTIL  condition
```

The condition will be a test of the outcome of the action *within* the REPEAT ... UNTIL ... loop. This is an extremely valuable mechanism. For example, you may have met the need to terminate an 'INPUT' list with something like line 140 of the following program segment:

```
100 REM Input loop
110 I = 1
120 INPUT "Enter next name: " N$
130 IF N$ <> "ZZZZ" THEN name$(I) = N$ : I = I + 1
140 IF N$ <> "ZZZZ" THEN 120
```

It's awkward, but so far unavoidable. But with REPEAT ... UNTIL ... a much neater solution is available.

Example 1
Input numbers into an array, terminating the inputting with the dummy value -9999.

Solution

```
10 REM REPEAT
20
30 REM REPEAT ... UNTIL demonstration
40 REM Inputs data until terminating value
```

```
 50
 60 DIM A%(50)
 70
 80 I% = 0
 90 REPEAT
100   I% = I% + 1
110   INPUT "Next number " A%(I%)
120   UNTIL A%(I%) = -9999 OR I% = 50
130
140 IF  A%(I%) = -9999 THEN I% = I% - 1
150 PRINT ' "List entered: contains "; I%  " numbers"
```

Program 2.1

K Type and run program 2.1.

You will notice that `REPEAT ... UNTIL ...` is one of the statements indented in listings in BBC BASIC by `LISTO` options 4 to 7. (Programs in this book are shown in LISTO 7).

SAQ 1

Devise a `REPEAT ... UNTIL ...` loop to display a table of squares of numbers up to a maximum square value of 1000.

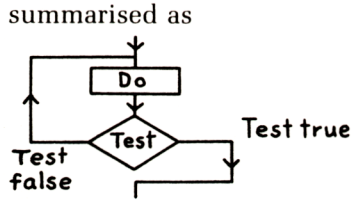
There are three points to notice about `REPEAT ... UNTIL ...`

- The result is tested on exit so you must make allowances where that would cause trouble, e.g. by repeating something once too many times. Alternatively, you can use a `WHILE` loop (see below).
- Because of the above, a `REPEAT ... UNTIL ...` loop is always executed at least once (just as is a `FOR ... NEXT ...` loop).
- You can have one-line `REPEAT ... UNTIL ...` loops if needed. It is perhaps to be avoided from a stylistic point of view – see the rules for good style in Unit 1. However, Program 2.2 demonstrates how it can be done.

```
10 REM REPEAT2
20 REM One line REPEAT ...UNTIL ...
30
40 I% = 0
50 REPEAT : I% = I% + 1 : UNTIL I% = 25
60
70 PRINT I%
```

Program 2.2

The `REPEAT` loop can be summarised as



WHILE

In addition to REPEAT, some BASICs offer WHILE ... ENDWHILE. This is not available in BBC BASIC, so we will describe how to construct it from a modified REPEAT loop. Indeed WHILE is only a REPEAT loop that tests at the beginning instead of at the end. It is particularly useful where we need a repeating loop but one which may not be executed at all in certain circumstances.

Example 2

Construct a loop to pad out a string with asterisks until its length is 10.

Solution

The routine must go on padding until the length is 10. However, the length may be 10 (or more) at the start of the loop so we could only use an ordinary REPEAT loop with care since all REPEAT loops are executed at least once. What we need is something which works like this:

```
WHILE length < 10

string = string + "*"

ENDWHILE
```

We can construct this using REPEAT ... UNTIL ... as in Program 2.3. (WHILE loops can also be constructed in BASICs without REPEAT ... UNTIL ... See Appendix 3 for more details).

```
10 REM WHILE
20
30 INPUT "Your name " name$
40
50 REM WHILE LOOP
60 REPEAT
70   IF LEN(name$) < 20 THEN name$ = name$ + "*"
80   UNTIL LEN(name$) >= 20
90
100 PRINT name$
```

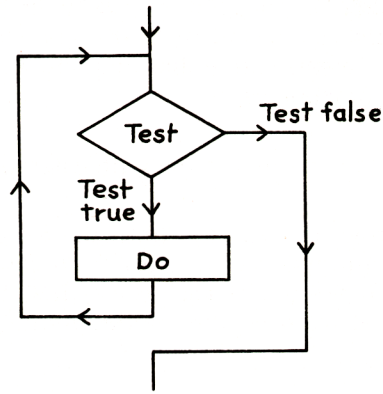
Program 2.3

If the exit test is complicated, you can avoid repeating it (once to control the action and once to exit) by using a flag e.g.

```
OK = FALSE
REPEAT
  IF <test> fails THEN ok = TRUE
  IF NOT ok THEN <action>
  UNTIL ok
```

WHILE is useful if your dialect of BASIC has it. However, since it is so close to REPEAT we shall not specifically mention WHILE loops again in

this book. They will just appear as REPEAT loops with the first action inside the loop being a test to decide whether or not the rest of the loop is to be executed. The WHILE loop can be summarised as:



Loops compared

You have now met the three types of loop that occur in programming:

Type of loop	Expressed in BASIC as:
Repeat a fixed number of times	FOR ... NEXT
Repeat until a condition is/is not fulfilled	REPEAT ... UNTIL ...
Repeat while a condition is/is not fulfilled	WHILE ... ENDWHILE

Any one of these loops can be pushed into doing the job of any of the others, but if you want your programs to be fast and elegant, you need to use the right loop for each job. The following rules should help in deciding which loop to use when (assuming that your BASIC supports all three):

- ☐ If the number of times a repetition is to occur will be known on entry to a loop, always use a FOR ... NEXT loop. It is faster than using one of the other loops for a fixed number of repetitions since the other forms are slowed down by the counter that you will have to build into them.
- ☐ If the condition that will terminate the repetition will be determined by the repeated action itself (i.e. the condition cannot be known on entry to the loop), then use REPEAT ... UNTIL.
- ☐ If the condition for exit from the loop could be satisfied before entry, then use WHILE ....

The relative speed advantage of FOR ... NEXT over REPEAT ... UNTIL ... for a fixed number of repetitions is demonstrated by Program 2.4. This also allows you to compare the speed of the two types of loop when you

use integer counters in place of floating point counters. In the case of FOR ... NEXT, the integer counter is 65 percent faster than the floating point counter. In the case of REPEAT ... UNTIL ..., the integer counter is 37 percent faster. (The program makes use of TIME which we have not yet explained. It is covered in Unit 3 in more detail).

```
 10 REM Comparison of loop speeds
 20
 30 REM FOR ... NEXT with floating point index
 40
 50 TIME = 0
 60 FOR I = 1 TO 5000
 70   NEXT
 80 time = TIME
 90 PRINT "FOR ... NEXT with FP index" TAB(35) ;
    time / 100 " secs"
100
110 REM FOR ... NEXT with integer index
120 TIME = 0
130 FOR I% = 1 TO 5000
140   NEXT
150 time = TIME
160 PRINT "FOR ... NEXT with integer index " TAB(35) ;
    time / 100 " secs"
170
180 REM REPEAT loop with floating point index
190 TIME = 0
200 I = 1
210 REPEAT
220   I = I + 1
230   UNTIL I = 5000
240 time = TIME
250 PRINT "REPEAT loop with FP index" TAB(35) ;
    time / 100 " secs"
260
270 REM REPEAT with integer index
280 TIME = 0
290 I% = 1
300 REPEAT
310   I% = I% + 1
320   UNTIL I% = 5000
330 time = TIME
340 PRINT "REPEAT loop with integer index" TAB(35) ;
    time / 100 " secs"
```

Program 2.4

```
Test run
>RUN

FOR ... NEXT with FP index      2.52 secs.
FOR ... NEXT with integer index   0.88 secs.
REPEAT loop with FP index       13.76 secs.
REPEAT loop with integer index    8.62 secs.
```

NEXT versus NEXT I

In Program 2.4, the NEXT statements were used without a variable, i.e. NEXT instead of NEXT I or NEXT I%. This omission of the variable in a NEXT statement speeds up the processing. If you insert the variable I into line 70 and I% into line 140, the times for the two FOR ... NEXT loops become 3.24 and 1.11 seconds respectively. When speed is critical you should therefore omit the variable from the NEXT statement. You may also wish to omit it on other occasions but in this book we have left the variable in so as to maximise the clarity of the loops.

2.2 IF ... THEN ... revisited

Before we look at more powerful decision-making statements, we will first look at the power of the BBC IF ... THEN ... statement. In *30 Hour BASIC*, we only used this to direct a program to a line number, e.g.

```
IF A > B THEN 260
```

In BBC BASIC, we can put any valid command that we like after THEN, including multiple statements, for example:

```
10 INPUT A
20 INPUT B
30
40 IF A > B THEN PRINT "First number is larger" :
   GOSUB 1000
50 IF A = B THEN PRINT "Numbers are equal" :
   GOSUB 2000
60 IF A < B THEN PRINT "Second number is bigger"
70
80 REM More program here
```

Program 2.5

Everything after a THEN and on the same line as that THEN is only executed when the IF condition is true. In

```
10 INPUT A
20
30 IF A < 0 THEN A = 0 : PRINT "Count that as zero" :
   GOSUB 500
40
50 PRINT A
```

Program 2.6

the statements on line 30 after THEN are not executed if A is equal to or greater than 0.

SAQ 2

Rewrite Program 2.7 using print statements on the IF ... THEN ... lines.

```
10 INPUT "First " first
20 INPUT "Second " second
30 IF first < second THEN 70
40 IF first = second THEN 60
50 PRINT "First is greater" : GOTO 80
60 PRINT "First and second are equal" : GOTO 80
70 PRINT "First is less than second"
80 END
```

Program 2.7

This small extension to IF ... THEN ... takes us some way towards the elimination of GOTO where it is used to jump a line or two in local decision making. Indeed all GOTOs of this type can be removed by either (a) test reversal or (b) duplicating the test in mutually exclusive forms:

Removal of GOTO by test reversal:

```
100 IF A < B THEN 120
110 PRINT "Something"
120 ...
```

becomes

```
100 IF A >= B THEN PRINT "Something"
120 ...
```

(Note that the reverse of '<' is '>=' and that the reverse of '>' is '<=').

Removal of GOTO by duplicating the test:

```
100 IF A = B THEN 130
110 PRINT "A not equal to B"
120 GOTO 140
130 PRINT "A equals B"
140 ...
```

becomes

```
100 IF A = B THEN PRINT "A equals B"
110 IF A <> B THEN PRINT "A not equal to B"
```

IF ... THEN ... ELSE ...

This statement works in the same way as IF ... THEN ... but allows you the extra facility of stating what should happen when the IF condition is not fulfilled. So the statement has the form:

`IF` (condition) `THEN` (BASIC statement) `ELSE` (BASIC statement)
(or a line number) (or a line number)

Programs 2.8 and 2.9 are examples of the use of `IF ... THEN ... ELSE ...`

```
10 REM DISCOUNT CALCULATOR
20
30 REM rate = discount rate
40
50 INPUT "Total? " total
60 IF total < 1000 THEN rate = .15  ELSE rate = .25
65
70 PRINT ' "Discounted total    " ; total * (1 - rate)
```

```
>RUN
Total? 85

Discounted total    72.25

>RUN
Total? 1345

Discounted total    1008.75
```

Program 2.8

```
10 REM Library fine calculator
20
30 REM days = days since loan started
40 REM fine = Fine due (pence)
50 REM 21 days are free
60 REM Fine for first 7 days overdue is 5p per day
70 REM Fine thereafter is 7p per day
80
90 REPEAT
100   PRINT ' "Next book"
110
120   INPUT "Days borrowed? " days
130
140   IF days > 28 THEN fine = 35 + 7 * (days - 28) ELSE
    IF days >  21 THEN fine = 5 * (days - 21)
150   IF days > 21 THEN PRINT "Fine due     " ;
    fine "p" ELSE PRINT "Nothing to pay."
160
170   UNTIL FALSE
```

Program 2.9

Program 2.9 used UNTIL FALSE to create a REPEAT loop that goes on indefinitely. Exit from the loop only occurs when the condition tested is TRUE, which FALSE never can be, so the loop goes on forever.

K Type and run Programs 2.8 and 2.9 and check that the decision-making routines behave as you would expect them to.

A potential problem with ELSE is highlighted by Program 2.9: the creation of long, confusing conditions. Sometimes it is better to avoid ELSE simply so as to make the logic easier to follow. For example, in Program 2.9 it is clearer to break line 140 up into two lines:

```
140 IF days > 21 THEN fine = (days - 21) * 5
145 IF days > 28 THEN fine = fine + (days - 28) * 2
```

Notice a crucial difference between IF ... THEN ... and IF ... THEN ... ELSE .... In IF ... THEN ..., nothing on the program line after THEN is executed when the IF condition is false. But with IF ... THEN ... ELSE ..., the line executes either what follows THEN or what follows ELSE. Thus

IF (condition) THEN (action 1) ELSE (action 2)

Executed if condition is true — Executed if condition is false

SAQ 3

What is the result of inputting (a) 8, 3, 9, 4 and (b) 2, 6, 18, 3 and (c) 4, 2, 10, 4 into:

```
10 INPUT A, B, C, D
20 IF A + B > C - D THEN Q = 2 * A + B ELSE Q = C - D
30 PRINT Q
```

SAQ 4

Rewrite Program 2.29 (in answer to SAQ 2) using ELSE to eliminate the need for lines 50 and 60.

Multiple IF ... THENs

The action in IF (condition) THEN (action) may itself be an IF ... THEN ... statement. You can cascade IF ... THENs one after another as in:

IF (condition 1) THEN IF (condition 2) THEN IF (condition 3) THEN (action)

In such a case, whether or not the final action is to be executed depends on a series of tests, each of which is only applied if the previous test is proved true. A practical example would be:

```
IF valid THEN IF weight > 400 THEN GOSUB 2000
```

We can also add an ELSE onto the end of a multiple IF ... THEN ... as in

```
IF valid THEN IF weight > 400 THEN GOSUB 2000 ELSE GOSUB 3000
```

Notice that the final ELSE action will be executed if *any one* of the IF tests fails:

```
 10 REM Multiple IF's
 20
 30 condition1 = TRUE : condition2 = TRUE
 40 PROCtest
 50
 60 condition1 = TRUE : condition2 = FALSE
 70 PROCtest
 80
 90 condition1 = FALSE : condition2 = TRUE
100 PROCtest
110
120 condition1 = FALSE : condition2 = FALSE
130 PROCtest
140
150 END
160
170 DEF PROCtest
180 IF condition1 THEN IF condition2 THEN PRINT
    "Action1" ELSE  PRINT "Else action"
190 ENDPROC


>RUN
Action1
Else action
Else action
Else action
```

Program 2.10

As a general rule, these cascaded tests should be avoided if possible, since they are hard to follow and so mistakes in their structure are easily made. We have used several in this book prior to introducing procedures. Once we are able to use procedures, we can usually make long tests simpler to the human eye.

2.3 AND, OR, NOT and EOR

AND

So far, the conditions which we have tested have had to be single items: e.g.

Test	Condition
`IF A <> 3 THEN 500`	`A <> 3`
`IF A = B THEN PRINT "OK"`	`A = B`

However, there are many occasions when the condition for something to happen is a great deal more complex. For example, to enter a police force one might have to be both over 6 feet in height and under 30 years of age. So if height is `height` (ins) and age is `age` (years) then the condition for entry is

```
height >= 72 AND age < 30.
```

The use of `AND` in this way turns the two conditions into a single condition which can go into a BASIC statement. Program 2.11 demonstrates this.

```
 10 REM AND demonstration
 20
 30 REPEAT
 40   PRINT ' "Next candidate"
 50   INPUT "Height (ins)? " height
 60   INPUT "Age (years)? " age
 70   PRINT
 80
 90   IF height >= 72  AND  age <  30 THEN  PRINT "Candidate
      accepted" ELSE PRINT "Candidate outside limits"
100
110   UNTIL FALSE

>RUN

Next candidate
Height (ins)? 73
Age (years)? 24

Candidate accepted

Next candidate
Height (ins)? 72
Age (years)? 24

Candidate accepted
```

Program 2.11

K Type and run Program 2.11.

The test for Program 2.11 was 'IF height >= 72 AND age < 30'. It could have been written as 'IF height >= 72 THEN IF age < 30 ...'. Superficially they appear to be the same – indeed they produce the same result in Program 2.11. But there are circumstances where the cascaded IF condition must be used and not the AND condition.

Cascaded IFs are needed where the use of AND could cause an evaluation error such as a division by zero, e.g.

```
IF A > -50 AND B < 12 THEN C = B / A
```

produces an error when A = 0, even though the IF condition is satisfied. Here you have to use:

```
IF A <> 0 THEN IF A > -50 AND B < 12 THEN C = B/A.
```

OR

OR works in a similar way to AND but has a convention that needs to be remembered:

```
IF A = 3  OR  B = 8 THEN  300
```

will direct a program to line 300 when:

A = 3
or B = 8
or A = 3 *and* B = 8.

So the BASIC OR is not like the 'or' of 'You can have cake or ice cream' where it is implied that you can't have both. The BASIC OR means cake or ice cream or both. For this reason it is called the inclusive OR, i.e. it includes the case when both conditions are met.

You can use it to cover several different circumstances, e.g. in Program 2.12 the single test in line 100 checks both age and number of years driving without a claim.

```
10 REM OR demonstration
20 REM Entitlement to no-claim bonus
30
40 REPEAT
50   PRINT ' "Next claimant"
60
70   INPUT "Age? " age
80   INPUT "No. of years driving without claim? " years
90
```

```
100     IF age  < 24  OR years  < 3  THEN PRINT  "Not
        entitled"  ELSE PRINT "Entitled to bonus"
110
120   UNTIL FALSE

>RUN

Next claimant
Age? 34
No. of years driving without claim? 2
Not entitled

Next claimant
Age? 23
No. of years driving without claim? 4
Not entitled

Next claimant
Age? 45
No. of years driving without claim? 3
Entitled to bonus
```

Program 2.12

K Type and run Program 2.12.

In the next example, Program 2.13, three ORs are run together. To check whether or not the three lengths specified by the user can form a triangle, we need to check that any two sides are greater in length than the third. So we need to test whether:

S1 + S2 < = S3
S1 + S3 < = S2
S2 + S3 < = S1

This is all done in one line (line 120).

```
 10 REM Draw a triangle
 20
 30 REPEAT
 40
 50   PRINT
 60   possible = TRUE
 70
 80   INPUT "First side? " S1
 90   INPUT "Second side? " S2
100   INPUT "Third side? " S3
```

```
110
120   IF S1  + S2  <= S3  OR S1  + S3  <= S2  OR S2  + S3  <=
    S1  THEN PRINT "Impossible triangle" : possible = FALSE
130
140   UNTIL possible
150
160 REM More program here

>RUN

First side? 3
Second side? 5
Third side? 9
Impossible triangle
```

Program 2.13

SAQ 5

What will be the output from these program segments when the variables have the values shown?

(a) `IF A = 7 AND B <> 2 THEN PRINT "X" ELSE PRINT "Y"`
when `A = 7` and `B = 2`

(b) `IF A > 5 AND B < 4 THEN PRINT "X" ELSE PRINT "Y"`
when `A = 6` and `B = 2`

(c) `IF A < 1 OR B > 1 THEN PRINT "X" ELSE PRINT "Y"`
when `A = -2` and `B = -1`

(d) `IF A <> B AND B > 3 THEN PRINT "X" ELSE PRINT "Y"`
when `A = 5` and `B = 2`

NOT

The next new word to put into `IF` tests is `NOT`. It is used to produce the logical opposite of a condition.

For example, you might have the condition `A = 3 AND B = 5` and you later want to do something when this condition isn't true. With `NOT` you can simply write:

```
IF NOT (A = 3 AND B = 5) THEN ...
```

The position of the brackets is very important since `NOT (A = 3 AND B = 5)` is different from `NOT (A = 3) AND B = 5`. In Program 2.14, `NOT` is used to test whether or not three conditions are simultaneously met.

```
10 REM NOT demonstration
20
30 REM Mortgage checker
40
```

```
 50 REPEAT
 60
 70   accepted = TRUE
 80
 90   INPUT "House post 1914? (Y/N) " young$
100   INPUT "Brick/stone? (Y/N)      " solid$
110   INPUT "In good repair? (Y/N)   " sound$
120
130   IF NOT (young$ = "Y" AND solid$ = "Y"  AND sound$ = "Y")
      THEN accepted = FALSE
140   IF accepted THEN PRINT "Accepted" ELSE PRINT "Not
      accepted"
150
160   UNTIL FALSE

>RUN
House post 1914? (Y/N) Y
Brick/stone? (Y/N)     Y
In good repair? (Y/N)  Y
Accepted
House post 1914? (Y/N) Y
Brick/stone? (Y/N)     N
In good repair? (Y/N)  Y
Not accepted
```

Program 2.14

Remember that NOT is not a Boolean operation but a mathematical one. You need therefore to restrict using NOT to expressions which are either -1 (= TRUE) or 0 (= FALSE).

Finally, you can combine AND, OR, EOR and NOT in limitless ways to build up multiple tests:

Example 3
In a (fictional) social security system, payments are made according to the following rules:

No payment if:

0 children and income greater than £40
1 child and income greater than £55
2 children and income greater than £80

£20 payment if:

0 children and income is £40 or less
1 child and income between £41 and £55
2 children and income between £61 and £80

£30 payment if:

1 child and income £40 or less
2 children and income £50 or less

For the sake of simplicity we have not extended the system to families with more children. Line 160 (Program 2.15) traps cases not catered for by the program. The rules themselves are applied in lines 210, 240 and 250.

```
 10 REM AND and OR demonstration
 20
 30 REM Social security check
 40
 50 REPEAT
 60
 70   claim = 0
 80
 90   REPEAT
100     PRINT ' "Next client"
110     INPUT "Weekly income? " income
120     INPUT "No of dependants? " dependants%
130
140     IF dependants% < 0 OR dependants% > 2 THEN PRINT "Out
        of range"
150
160     UNTIL dependants% >= 0 AND dependants% <= 2
170
180   done = FALSE
190
200   REM Remove clients with no claim
210   IF (dependants% = 0 AND income > 40) OR (dependants% = 1
      AND  income > 55) OR (dependants% = 2 AND income > 80)
      THEN claim = 0 : done = TRUE
220
230   REM Evaluate valid claims
240   IF NOT done THEN IF (dependants% = 0) OR (dependants% = 1
      AND income > 40) OR (dependants% = 2 AND income > 60)
      THEN claim = 20 : done = TRUE
250   IF NOT done THEN IF (dependants% = 1) OR (dependants% = 2
      AND income > 50) THEN claim = 30
260
270   PRINT "Payment due = " ; claim
280
290   UNTIL FALSE
>RUN
```

```
Next client
Weekly income? 45
No of dependants? 0
Payment due = 0

Next client
Weekly income? 56
No of dependants? 2
Payment due = 30
```

Program 2.15

EOR

As we have said, `A OR B` means '`A` or `B` or both'. `A EOR B` means '`A` or `B` but not both'. It is called the *exclusive* `OR` because it excludes the case where both `A` and `B` are true (and also where `A` and `B` are both false).

Program 2.16 shows `EOR` being used to test whether one, and only one, of two random numbers generated by a loop is greater than 500.

```
  10 REM EOR demonstration
  20
  30 PRINT "first", "second"
  40
  50 REPEAT
  60   first = RND(1000)
  70   second = RND(1000)
  80   PRINT ; first, ; second
 100   UNTIL first > 500 EOR second > 500

>RUN

first     second
147       289
682       980
205       727
```

Program 2.16

Expression priority

We have used brackets in many of the multiple conditions which we have set up, e.g.

```
  IF (dependants% = 0 AND income > 40) OR (dependants% = 1 AND
income > 55)
      OR (dependants% = 2 AND income > 80)
```

We inserted these for clarity but, in this particular case, the computer does not need them. Multiple conditions are evaluated by the BBC

Micro in the following priority order:

First	NOT
Next	AND
Next	OR and EOR with equal priority

Applying this priority order to the statement above, you can see that the parts we put in brackets:

```
(dependants = 0 AND income > 40)
(dependants = 1 AND income > 55)
(dependants = 2 AND income > 80)
```

would in any case have been evaluated first since AND has a higher priority than OR. (Since there is no NOT in this example, AND is evaluated first).

SAQ 6

Remove any brackets in the following which are not essential to the evaluation of the tests as set out:

(a) NOT(A = B AND (C = D AND E > F)) OR (X = Y AND L < M)
(b) (A = B AND (C = D OR X = Y)) AND (NOT (P = Q OR R > S))
(c) A > B AND NOT ((C = D OR E = F) AND X = Y)
(d) (A = B AND C = D) OR (X = Y AND L = M) AND NOT (P = Q OR D = F)
(e) NOT (((A = B OR C = D) AND (E = F OR G = H) OR (X = Y)) AND P = Q)

SAQ 7

On a rail system there are two types of ticket: full-fare and half-fare. Half-fare tickets are available to everyone on Sundays. On other days, half-fare tickets are available outside rush hours to pensioners and to people under 14. Pension age for men is 65 and for women is 60. Design a program in which, for each traveller, the user enters information in the following form:

```
Sunday travel?  (Y/N)
Rush hour travel?  (Y/N)
Age? (years)
Sex? (M/F)
```

The program will then print out 'Half-fare ticket' or 'Full-fare ticket' as appropriate. Ensure that your program will not accept invalid data.

2.4 ON ... GOTO ...

So far you have met IF ... THEN ... which allows you to switch program execution to a line other than the next line in the program sequence and IF ... THEN ... ELSE ... which allows you to switch to a choice of two lines. ON is a multi-way program switch, redirecting the program according to the value of what follows ON.

```
120 ON X GOTO 150, 300, 700
```

will move execution to line 150 if X = 1, to line 300 if X = 2 and to line 700 if X = 3. Notice that ON is only able to make decisions on consecutive values of X starting at 1. If an ON variable has *consecutive* values but they don't start at 1, you can still use ON as follows:

```
100 REM Y has values 8, 9, 10. Y - 7 produces values 1, 2, 3
110 ON Y - 7 GOTO 150, 300, 700
```

There are three points to remember whenever you use ON:

- ☐ The values of your variable (adjusted if need be) must be consecutive and start at 1.
- ☐ The destination line numbers must exist as actual lines.
- ☐ Any non-integer value of the variable will be treated as the next smaller integer value.
- ☐ Your program must prevent your variable taking a value greater than the number of options in the ON list. If you fail to do this, you get the error message ON RANGE AT LINE ... and your program halts.

There are various methods of trapping invalid values of your ON variable. Here are some for a case where values 1, 2 and 3 are permitted:

By checking the variable

```
 90 IF X < 1 OR X > 3 THEN [trap]
100 ON X GOTO 200, 500, 400
```

By use of ON ... GOTO ... ELSE ...
ELSE can be tacked onto ON ... GOTO ... just as it can with IF ... THEN ... One use for ELSE is to trap invalid ON values:

```
 90 INPUT X
100 ON X GOTO 200, 500, 400 ELSE 90
```

By use of ON ERROR

```
 80 ON ERROR GOTO 90
 90 INPUT X
100 ON X GOTO 200, 500, 400
110 ON ERROR OFF
```

(ON ERROR is explained fully in Section 2.5. As you will see, lines 200, 500 and 400 would have to re-set ON ERROR to a new destination).

So, a generalised example of the use of ON ... GOTO ... would be:

```
10 REM ON demonstration
20
30 REM ON directs program to lines in list
40 REM depending on value of ON variable
50
```

```
 60 INPUT X
 65 REM Note ON list need not be in line number order
 70 ON X GOTO 200, 800, 500 ELSE 60
 80
 90
200 REM Program option 1 here
210 REM Selected when X = 1
220
500 REM Program option 3 here
510 REM Selected when X = 3
520
800 REM Program option 2 here
810 REM Selected when X = 2
820
```

Program 2.17

However, as you will see in Unit 5, the use of ON ... GOTO ... is to be avoided on stylistic grounds.

SAQ 8

What are the mistakes in this program extract?

```
250 PRINT "Enter size 2, 3, 4 or 6 " size
260 ON size GOTO 300, 700, 260
```

ON ... GOSUB ...

ON ... GOSUB ... and ON ... GOSUB ... ELSE ... work in the same way as ON ... GOTO ... with the following additional points:

- ON ... GOSUB ... switches execution to GOSUBs so that GOSUBs must exist at all the line numbers that you switch to.
- Once any one of the GOSUBs is executed, the program will return to the line after the ON line, so you need to make sure that the program continues with a section that is supposed to follow after the particular ON option that has been executed.
- The first release of BBC BASIC has a bug when ELSE is used after ON ... GOTO ... or ON ... GOSUB ... If you have this version of the BASIC, you will need to avoid using ELSE in this way. (See *User Guide*, page 309).

ON ... GOTO ... versus IF

ON ... GOTO ... looks attractive since it gives us an easy way to transfer flow to different sections of a program. Stylistically, however, it is best avoided, since (a) it allows the opportunity of errors in line numbers and (b) it is necessary to refer to other parts of the program to discover what is happening at each of the destinations. As you will see in this

book, we stick to cascaded `IF`s (e.g. Program 5.8) which are much easier to read.

BBC BASIC also permits 'computed' `ON ... GOTO`s, e.g.

```
ON A GOTO 1000 + A * 10
```

This type of construction is to be avoided at all costs, since renumbering cannot cope with the new value to be assigned to `A * 10` and the program will not work as intended.

2.5 Error trapping and reporting

Automatic error messages

You will almost certainly have noticed that even when your programs are free of syntax errors and can do exactly what you intended, it is still possible for them to go wrong when the user does something which you overlooked. The result is that the computer stops and gives you an error message. In a short program, that may not matter but in a long program the result can be infuriating, e.g. if the user has entered 30 names and addresses for holding in the computer's memory, they will all be lost to him when the program 'crashes'. (The data is still in the variables, and accessible to a programmer, but hardly to a general user). Fortunately, BASIC has systems for trapping these errors and allowing you, the programmer, to decide what should happen if they occur. But to do that you have to have some idea of what the errors might be. Let's look first at the commonest ones.

Programs 2.18 to 2.21 demonstrate in turn the computer's responses to:

(a) attempting to divide by zero
(b) attempting to exceed the capacity of an array
(c) pressing `ESCAPE` during a run of the program
(d) attempting to put string data into a numerical variable.

In each case the computer stops and prints the error message shown.

(a) Attempting to divide by zero

```
 10 REM Error demonstration
 20 REM Division by zero
 30
 40 REPEAT
 50
 60   INPUT "First number " A
 70   INPUT "Second number " B
 80
 90   PRINT A " / " ; B " = " ; A / B '
100
110   UNTIL FALSE
```

```
>RUN
First number 486
Second number 45
       486 / 45 = 10.8

First number 78
Second number 0
        78 / 0 =
Division by zero at line 90          Error message
```

Program 2.18

K Type and run Program 2.18.

(b) Attempting to exceed the capacity of an array

```
  10 REM Error demonstration
  20 REM Exceeding capacity of an array
  30
  40 DIM A(5)
  50
  60 PRINT "End entries with -9999"
  70
  80 I% = 0
  90 REPEAT
 100
 110   INPUT "Next number " N
 120
 130   IF N <> -9999 THEN A(I%) = N : I% = I% + 1
 140
 150   UNTIL N = -9999
 160
 170 PRINT "Entries completed"

>RUN
End entries with -9999
Next number 45
Next number 76
Next number 87
Next number 32
Next number 12
Next number 39
Next number 92

Subscript at line 130                Error message
```

Program 2.19

K Type and run Program 2.19.

(c) Pressing escape during a run

```
 10 REM Error demonstration
 20 REM Effect of escape
 30
 40 DIM A(5)
 50
 60 I% = 1
 70 REPEAT
 80
 90   INPUT "Next number " A(I%)
100
110   I% = I% + 1
120
130   UNTIL I% = 6
140
150 PRINT "Entries ended"

>RUN
Next number 34
Next number 65
Next number
Escape at line 90            Error message: user pressed escape
```

Program 2.20

K Type and run Program 2.20.

(d) Attempting to put a string into a numerical variable

```
 10 REM Error demonstration
 20 REM Mixing strings and numbers
 30
 40 DIM N(50)
 50
 60 REM Read in data
 70
 80 I% = 1
 90 REPEAT
100
110   READ N
120   PRINT N
130
140   N(I%) = N
150   I% = I% + 1
160
170   UNTIL N = - 9999
```

```
180
190 PRINT "Entries ended"
200
210 END
220
230 REM DATA list with string where program expects
    a numeric variable
240 DATA 34, 56, 34, "SUNDAY", 56, -9999
```

Quotation marks are not usually needed in a DATA line but we have put them here to force this particular error message.

```
>RUN
        34
        56
        34

Type mismatch at line 110
```

Error message. Computer expected a number and found a string.

Program 2.21

K Type and run Program 2.21.

Controlled error reporting

The BBC Micro has four options available to the programmer to trap errors and still keep the program running. We shall look at each of these in turn.

ON ERROR causes the computer to execute the next command after ON ERROR whenever an error occurs. Program execution then continues at the next statement after ON ERROR. You can make use of this in many ways including:

```
ON ERROR GOTO ...
ON ERROR GOSUB ...
ON ERROR REPORT
ON ERROR PROC ...
```

We now discuss some of these in more detail.

Method 1 ON ERROR GOTO

The simplest option is `ON ERROR GOTO`. This returns the program section covered by your `ON ERROR` statement to whatever line you choose to specify. In a very simple program, it's fairly obvious where you want to return the program to. For example, in Program 2.18, you will want to return the user to line 40. So you put an `ON ERROR` line before the error could occur, e.g.

```
                                35 ON ERROR GOTO 40

   10 REM Error demonstration
   20 REM Division by zero
   30
   35 ON ERROR GOTO 40
   40 REPEAT
   50
   60   INPUT "First number " A
   70   IF A = -9999 THEN END
   80   INPUT "Second number " B
   90
  100   PRINT A " / " ; B " = " ; A / B '
  110
  120   UNTIL FALSE

>RUN

First number 43
Second number 0
        43 / 0 =        Division by zero error trapped by ON ERROR
                        Computer jumps to line 40
First number 43
Second number 56
        43 / 56 = 0.767857143


First number -9999
```

Program 2.22

K Type and run Program 2.22.

As you can see, the result completely solves the problem – the program keeps on running whatever the user does – until he presses **BREAK**.

In a larger program you might have several error traps, each returning the program to the appropriate place. This is achieved by use of **ON ERROR ...** together with **ON ERROR OFF**, as follows:

```
 90
100 ON ERROR GOTO 110      All errors in this gap switch
110                )       control to line 110
120                )
                   )

190 ON ERROR OFF           Cancels the last ON ERROR
200 ON ERROR GOTO 210      All errors in this gap switch
210                )       control back to 210
220                )
                   )

300  ON ERROR OFF          Cancels the last ON ERROR
```

Method 2 ON ERROR GOSUB

This behaves in the same way as **ON ERROR GOTO** except that you send the program to a subroutine which copes with the error. Once the subroutine has been executed, the subroutine will return execution to the next statement after the **ON ERROR GOSUB** statement. You have, therefore, to think carefully about when and where to use this technique. An example of this will be given shortly.

Method 3 REPORT

Method 1 was fine but could possibly leave the user confused. For example, in Program 2.22, would the user know why his 0 entry had been refused? Well, in case he doesn't, we can get the computer to tell him why, at the same time as it returns control to wherever we want. All we do is replace

```
ON ERROR GOTO 40
```

by

```
ON ERROR REPORT
```

This automatically returns control to the **ON ERROR REPORT** line and the computer then prints out one of the error messages in BBC BASIC.

Program 2.23 shows the effect of ON ERROR REPORT on Program 2.22.

```
 10 REM Error demonstration
 20 REM Division by zero
 30
 35 ON ERROR REPORT
 40 REPEAT
 50
 60   INPUT ' "First number " A
 70   IF A = -9999 THEN END
 80   INPUT "Second number " B
 90
 90   PRINT A " / " ; B " = " ; A / B
100
110   UNTIL FALSE
```

Program 2.23

```
>RUN

First number 43
Second number 0
        43 / 0 =
Division by zero
First number 43
Second number 56
        43 / 56 = 0.767857143
```

Computer jumps here to ON ERROR REPORT line

Error report

K Type and run Program 2.23. Try the effect of pressing ESCAPE when it is running.

Method 4 Designing your own error messages

The final method is more tedious to use but gives you greater control. It makes use of the fact that the BBC Micro has a variable ERR which contains the number of the type of error that last occurred. You can use this to design your own error messages. But if there are a lot of possible error types, you need to design a lot of messages. Let's look at Program 2.23 again and put in an error trap with special error messages for pressing ESCAPE and attempting to divide by zero. The steps are:

- ☐ First look up the error numbers corresponding to these two errors (*User Guide*, Chapter 46). They are 18 for division by zero and 17 for pressing escape.

- ☐ Put an ON ERROR GOSUB line in the program (see method 2), e.g.

ON ERROR GOSUB 170.

☐ Store your error messages in the GOSUB. e.g.

```
190 IF ERR = 18 THEN PRINT "You cannot divide by zero."

200  IF ERR = 17 THEN PRINT "Do not press escape."
```

Program 2.24 is Program 2.23 modified to include these tailor-made error messages. (The program has been renumbered).

```
 10 REM Error demonstration
 20 REM Division by zero
 30
 40 ON ERROR GOSUB 170
 50
 60 REPEAT
 70
 80   INPUT "First number " A
 90   IF A = -9999  THEN END
100   INPUT "Second number " B
110
120   PRINT A " / " ; B " = " ; A / B '
130
140   UNTIL FALSE
150
160
170 REM Error messages subroutine
180
190 IF ERR = 18  THEN PRINT ' "You cannot divide by zero."
200 IF ERR = 17 THEN PRINT ' "Do not press ESCAPE."
210 PRINT "Please re-enter your data." '
220
230 RETURN
```

Program 2.24

```
>RUN
First number 45
Second number 0
        45 / 0 =
You cannot divide by zero.
Please re-enter your data.

First number 45
Second number  <------  User pressed ESCAPE here
Do not press ESCAPE.
Please re-enter your data.
```

```
First number 34
Second number 56
        34 / 56 = 0.607142857
```

K Type and run Program 2.24.

But where is the error?

The user of the program doesn't need to know where in the program the error occurred - he only needs to know what he did wrong and what to do next. But the designer of the program sometimes needs to know at what line the error occurred. If you are not using **ON ERROR**, you get the line numbers of errors automatically (see Programs 2.18 - 2.21) but the program always stops at the first error. If you are using **ON ERROR ...** or **REPORT**, the line numbers of the errors are lost. To get the numbers you can use the pseudo-variable **ERL**. It holds the line number where the last error occurred. If you want that line number to be printed out, include a message such as

```
PRINT "Error at line " ; ERL
```

at a suitable point in your error trap. This refinement has been added to Program 2.24 in order to give Program 2.25.

```
 10 REM Error demonstration
 20 REM Division by zero
 30
 35 ON ERROR GOSUB 150
 40 REPEAT
 50
 60   INPUT "First number " A
 70   IF A = -9999  THEN END
 80   INPUT "Second number " B
 90
100   PRINT A " / " ; B " = " ; A / B '
110
120   UNTIL FALSE
130
140
150 REM Error messages subroutine
160
165 PRINT '"Error at line " ; ERL
170 IF ERR = 18  THEN PRINT ' "You cannot divide by zero."
180 IF ERR = 17 THEN PRINT ' "Do not press ESCAPE."
190 PRINT "Please re-enter your data." '
200
210 RETURN
```

Program 2.25

```
>RUN
First number 45
Second number 0
        45 / 0 =
Error at line 55
You cannot divide by zero.
Please re-enter your data.

First number 45
Second number
Error at line 50
Do not press ESCAPE
Please re-enter your data.

First number 34
Second number 56
        34 / 56 = 0.607142857
```

K Type and run Program 2.25.

Multiple statements in an ON ERROR line

`ON ERROR` lines behave in the same way as `IF ... THEN ...` constructions in that the statements following `ON ERROR` on the same line are only executed if an error occurs. Program 2.25a demonstrates this.

```
 10 REM Multiple statements on an ON ERROR line
 20
 30 ON ERROR GOSUB 120 : PRINT "Statement after ON ERROR
    has been executed."
 40
 50 FOR I = 1 TO 10000
 60   I = I + 1
 70   NEXT I
 80 PRINT "Run ended."
 90
100 END
110
120 REM ON ERROR GOSUB
130 PRINT "Program is now in error trap."
140 PRINT "Press space bar to leave error trap."
150 REPEAT
160   UNTIL INKEY(0) <> TRUE
170 RETURN
```

Program 2.25a

K Run Program 2.25a. Whilst the loop (lines 50 to 70) is executing,

press Escape. This will take you into the error trap. Then press the space bar and you will find that the second statement on line 30 is executed. This statement will only be executed *after* a return from the error trap subroutine.

Because of this slightly unusual effect, we suggest that you try to avoid multiple-statement **ON ERROR** lines.

SAQ 9

Modify Program 2.26 to:

(a) trap errors only with **ON ERROR GOTO**;
(b) to trap errors and **REPORT** with line numbers at which the errors occur.

Note that you need three separate traps in this program for both (a) and (b).

```
 10 REM Data tabulation
 20 REM Inputs data and prints out in
 30 REM table to users specifications
 40
 50 DIM A$(101)
 60
 70 REM Input routine
 80
 90 CLS
100 I% = 0
110 REPEAT
120
130   I% = I% + 1
140
150   PRINT "Next item"
160   PRINT "End list with ZZZZ"
170   INPUT A$(I%)
180   UNTIL I% = 100 OR A$(I%) = "ZZZZ"
190
200 IF A$(I%) = "ZZZZ" THEN I% = I% - 1
210
220 REM Specify tabulation format
230
240 REPEAT
250
260   PRINT '' "Enter details for table."
270   INPUT "No. of rows " R%
280   INPUT "No. of columns " C%
290
300   REM Check table size
```

```
310   IF R% *  C% <  I% THEN  PRINT "Table  too small  for
      data.  Please re- enter."
320   UNTIL R% * C% >= I%
330
340 GOSUB 380
350
360 END
370
380 REM Table print out subroutine
390
400 CLS
410 FOR V% = 1 TO C%
420   FOR H% = 1 TO R%
430     PRINT TAB((H% - 1) * (40 / C%)) A$((V% - 1)
    * R% + H%) ;
440     NEXT H%
450   PRINT
460   NEXT V%
470 RETURN
```

Program 2.26

K Test your error trapping system for Program 2.26.

We have looked at ON ERROR GOSUB ... in some detail. From Unit 3 onwards you may prefer to use procedures instead of subroutines. In that case you would replace ON ERROR GOSUB ... with ON ERROR PROC ...

Three warnings

ON ERROR is a valuable mechanism but in three circumstances it can cause a great deal of trouble.

First, do *not* put any ON ERROR statements in a program which you are still developing and testing or typing in from a printed listing. They will hide your programming mistakes e.g. a syntax error such as a missing bracket may be obscured by an error trap. So write, perfect and test your program thoroughly before, as a final touch, adding error traps.

Second, be careful how you use ON ERROR to trap errors arising *within* a GOSUB.

If you do do this, the ON ERROR must re-direct the program out of the subroutine into the main linear part of the program. (In later issues of the BBC Micro operating system, it is possible to disable Escape in your programs if you wish to.) Meanwhile, Program 2.27 demonstrates what happens if you can't disable Escape, and you re-direct the program to inside a subroutine. (This point also applies to error trapping around procedures, which are introduced in the next unit.)

```
 20
 30 DIM N$(20)
 40
 50 I% = 0
 60 GOSUB 120
 70
 80 END
 90
100 REM Subroutine
110
120 ON ERROR GOTO 140
130
140 REPEAT
150   I% = I% + 1
160   INPUT "Next name " N$(I%)
170   UNTIL N$(I%) = "ZZZZ"
180
190 RETURN
```

Redirects error in subroutine to subroutine

Program 2.27

K Type Program 2.27 and enter data as follows:

(a) Enter FRED, JIM, MARY, ZZZZ. This should work as you would expect.

(b) Enter FRED, JIM; then press ESCAPE; then enter MARY and ZZZZ. What do you find?

(c) Press BREAK and key OLD to re-gain the program. Now change line 1000 to

```
1000 ON ERROR GOTO 10
```

and repeat (a) and (b).

What we have demonstrated is a disastrous feature of BBC BASIC. It makes error trapping near to impossible in well-structured programs. Every execution of 'ON ERROR' clears what is called the BASIC stack – a part of the memory that enables the computer to remember how to back-track on all the program's routings into loops, procedures, subroutines etc.

Third, if you have multi-statement lines, the error reports are hard to check. If line 500 has ten statements in it, 'Syntax error at line 500' still leaves you with a lot of checking to do. This is another good reason for sticking to our rule of 'one statement per line'.

Assignment B

1. Write a program which displays

1. Add
2. Subtract
3. Multiply
4. Divide
5. End

If the user selects 1 to 4, he can then enter two numbers and the arithmetic operation which he chooses is performed. Use `ON ... GOTO ... ELSE` and `ON ERROR` to deal with as many errors as possible. Try to anticipate any other invalid actions which the user might attempt.

2. A program contains a secret password, say, `R37BX2`, in an input routine. The user has to enter the password to access the next part of the program. He is allowed five attempts after which, if he fails, an alarm sounds. Devise a suitable input routine using `REPEAT ... UNTIL ...` You will find it useful to use one or more of `NOT`, `AND` and `OR` for part of the decision making. (Note: to sound an alarm on the BBC Micro, use `VDU 7`. How can you make `VDU 7`'s short beep into a longer one?)

Objectives of Unit 2

By the end of this Unit you should be able to use:

- ☐ `IF ... THEN ... ELSE ...`
- ☐ `AND`, `OR`, `EOR` and `NOT` either individually or in any combination
- ☐ `ON ... GOTO ...`
- ☐ `ON ... GOTO ... ELSE ...`
- ☐ `ON ... GOSUB ...`
- ☐ `ON ... GOSUB ... ELSE ...`
- ☐ `REPEAT ... UNTIL ...`
- ☐ `ON ERROR GOTO`
- ☐ `REPORT`
- ☐ `ERR`
- ☐ `ERL`

Answers to SAQs

SAQ 1

```
10 REM List squares up to 1000
20
30 I% = 0
40
50 REPEAT
```

```
 60
 70   I% = I% + 1
 80   square = I% * I%
 90   IF square < 1000 THEN PRINT ; I%, square
100
110   UNTIL square >= 1000
```

Program 2.28

SAQ 2

```
10 INPUT "First " first
20 INPUT "Second " second
30
40 IF first > second THEN PRINT "First is greater"
50 IF first < second THEN PRINT "First is less
   than second"
60 IF first = second THEN PRINT "First and second
   are equal"
70
80 END
```

Program 2.29

SAQ 3

(a) `A + B = 11` and `C - D = 5` so the condition is true. The `THEN` action is executed. So `Q = 19`.

(b) `A + B = 8` and `C - D = 15` so the condition is false. The `ELSE` action is executed. So `Q = 15`.

(c) `A + B = 6` and `C - D = 6`. Since the condition is `IF A + B > C - D`, it is false (6 is not greater than 6) so the `ELSE` is executed. `Q = 6`.

SAQ 4

```
10 INPUT "First " first
20 INPUT "Second " second
30
40 IF first > second THEN PRINT "First is greater" ELSE  IF
   first  < second THEN PRINT "First is less than second"
   ELSE PRINT "First and second are equal"
70
80 END
```

Program 2.30

SAQ 5

(a) Y (b) X (c) X (d) Y.

SAQ 6

We shall remove the unnecessary brackets, one set at a time:

(a) `NOT (A = B AND (C = D AND E>F)) OR (X = Y AND L<M)`

Not needed since we do not need to express a priority between two `AND`'s.

`NOT (A = B AND C = D AND E>F) OR (X = Y AND L<M)`

Not needed since `AND` already has a higher priority than `OR`.

`NOT (A = B AND C = D AND E>F) OR X = Y AND L<M`

These brackets are needed since `NOT A = B AND C = D AND E>F...` would be interpreted by the computer as `(NOT A = B) AND C = D...` This is because `NOT` has a higher priority than `AND`.

Hence the final result is

`NOT(A = B AND C = D AND E>F) OR X = Y AND L<M.`

(b) Again, we remove the unnecessary brackets one set at a time.

`(A = B AND (C = D OR X = Y)) AND (NOT(P = Q OR R > S))`

Not needed since `NOT` is higher priority than any other operation.

`(A = B AND (C = D OR X = Y)) AND NOT (P = Q OR R > S)`

Not needed since we don't need to express an order of priority between `AND`'s.

`A = B AND (C = D OR X = Y) AND NOT (P = Q OR R > S)`

These remaining brackets are needed since `A = B AND C = D OR X = Y` would be interpreted by the computer as `(A = B AND C = D) OR X = Y`

and

`NOT P = Q OR R > S` would be interpreted as `(NOT P = Q) OR R > S.`

So the final result is

`A = B AND (C = D OR X = Y) AND NOT (P = Q OR R > S).`

(c) None of the brackets can be removed.

(d) The first two pairs can be removed since `AND` takes priority over `OR`. So the expression can be written

`A = B AND C = D OR X = Y AND L = M AND NOT (P = Q OR D = F).`

(e) Only the brackets round `X = Y` can be removed.

SAQ 6 has been designed to help you to check whether you understand expression priority for `AND`, `OR` and `NOT`. It is not meant to imply that you will always wish to remove all superfluous brackets. Quite often it is useful to leave brackets in so as to add clarity, e.g. see line 300 of Program 2.31.

SAQ 7

```
 10 REM Ticket printer
 20 REM Prints full and half fare tickets
 30 REM According to rules in text
 40
 50 CLS
 60 REPEAT
 70
 80   PRINT "Next ticket" '
 90
100   REPEAT
110     INPUT "Sunday travel? (Y/N)     " day$
120     UNTIL day$ = "Y" OR day$ = "N"
130
140   REPEAT
150     INPUT "Rush hour travel? (Y/N) " time$
160     UNTIL time$ = "Y" OR time$ = "N"
170
180   REPEAT
190     INPUT "Age? (years)              " age
200     UNTIL age >= 0 AND age < 140
210
220   REPEAT
230     INPUT "Sex? (M/F)                " sex$
240     UNTIL sex$ = "M" OR sex$ = "F"
250
260   REM Decide which ticket
270
280   IF day$ = "Y" THEN sunday = TRUE ELSE sunday
      = FALSE
290   IF time$ = "Y" THEN rush hour = TRUE ELSE
      rush hour = FALSE
300   IF (age > 65 AND sex$ = "M") OR (age >60 AND sex$
      = "F") THEN concession = TRUE ELSE concession
      = FALSE
310
320   cheap = FALSE
330   IF concession THEN cheap = TRUE
340   IF rush hour THEN cheap = FALSE
```

```
350   IF sunday THEN cheap = TRUE
360   IF cheap THEN PRINT "    HALF FARE" ELSE PRINT "
          FULL FARE"
370
380   UNTIL FALSE
390   END
```

Program 2.31

SAQ 8

The mistakes are:

- ☐ Values of S are not consecutive – there is a gap between 4 and 6.
- ☐ Values do not start at 1 in ON line.
- ☐ ON line contains only 3 lines to switch to but four values of S have been offered to the user.
- ☐ The third option sends the ON line back to itself – this would lock the program into an endless loop.
- ☐ No error-trapping procedure.

SAQ 9

Add lines to Program 2.27 as follows:

(a)

`65 ON ERROR GOTO 90`	Traps errors in data input.
`215 ON ERROR GOTO 240`	Traps errors during table specification.
`335 ON ERROR GOTO 340`	Traps errors during display (i.e. stops the effect of the user pressing ESCAPE).

(b) This can be done in many ways. One possibility is to use a GOSUB :

```
440 REPORT
450 PRINT "Error at line " ; ERL
460 PRINT
470 RETURN
```

If you use this method then you would also need:

75, 185 and 325 as ON ERROR GOSUB 440

UNIT 3

Procedures and user-defined functions

3.1 Procedures

3.2 User-defined functions

3.3 TIME, MOD and DIV

3.4 TAB(X, Y)

Assignment C

Objectives of Unit 3

Answers to SAQs

3.1 Procedures

You have already met **GOSUB**. This allows you to use a subroutine many times in one program whilst only needing to write it out once. Subroutines are called by line number e.g. **GOSUB 2000**. At their simplest level, procedures are like subroutines except that they are called by name. In Program 3.1a, the dotted line is drawn by **GOSUB 100**. In Program 3.1b, it is drawn by **PROCdraw**.

```
 10 REM Use of GOSUB
 20
 30 MODE 7
 40 GOSUB 100
 50
 60 END
 70
 80 REM Subroutine
 90
100 FOR I = 0 TO 39
110   PRINT TAB( I ) "-" ;
120   NEXT I
130 PRINT
140
150 RETURN
```

Program 3.1a

```
 10 REM Use PROC
 20
 30 MODE 7
 40 PROCdraw
 50
 60 END
 70
 80 DEF PROCdraw
 90
100 FOR I = 0 TO 39
110   PRINT TAB( I ) "-" ;
120   NEXT I
130 PRINT
140
150 ENDPROC
```

Program 3.1b

A procedure has three parts to it:

	Example	Comparison with GOSUB
☐ The line which calls it	`PROCdraw`	`GOSUB 2000`
☐ The line which starts its definition	`DEF PROCline`	
☐ The line which ends its definition	`ENDPROC`	`RETURN`

Notice that

```
PROCdraw
```

is written without any spaces. '`PROC draw`' or '`PROCdr aw`' are not allowed. So for more complex names you can either run the words together (`PROCnextletter`) or use the underline character to separate them without introducing a space (`PROCnext_letter`), (i.e. procedure names follow the same rules as variable names).

We are using lower-case letters for the names of our procedures. We could use upper-case but we have chosen lower-case for clarity.

SAQ 1

Rewrite Program 3.2 using a procedure to replace the `GOSUB`.

```
 10 REM Input check
 20
 30 PRINT "Enter name (max 6 characters)"
 40 GOSUB 140
 50 name$ = A$
 60
 70 PRINT "Enter area code (max 6 characters)"
 80 GOSUB 140
 90 area$ = A$
100
110 END
120
130 REM Input suboutine
140 REPEAT
150
160   INPUT A$
170   IF LEN(A$) > 6 THEN PRINT "Too long. Please re-enter."
180   UNTIL LEN(A$) <= 6
190
200 RETURN
```

Program 3.2

K Type and run Program 3.2

Speed of procedures and subroutines

In the BBC Micro, calls to procedures are normally faster than calls to

subroutines. This is because subroutines have to be searched for (down the line numbers) every time that they are called. So the further down your program a subroutine is, the longer it takes to find. Procedures, on the other hand, are indexed in a special part of the memory and can be accessed at the same speed wherever they are in a program. If your BASIC only has subroutines and they are searched for down the program, then you need to plan where to place your subroutines. Subroutines that will be needed in parts of the program where speed is critical need to be placed at earlier line numbers than subroutines that perform less time-critical operations. The following activity allows you to compare the speeds of subroutines and procedures.

[K] Run Program 3.3 and see how long it takes to execute 5000 times the empty subroutine at line 110.

```
 10 REM Subroutine speed test
 20
 30 TIME = 0
 40 FOR I% = 1 TO 5000
 50   GOSUB 110
 60   NEXT I%
 70 PRINT TIME / 100
 80
 90 END
100
110 RETURN
```

Program 3.3

Our test run gave 3.67 seconds.

Now add new lines numbered 110 to 2990 (going up in tens), each just saying **REM**. Then change line 50 to **GOSUB 3000** and add **3000 RETURN** so that the subroutine is now 190 lines further down the program. Re-run the program and see how long it takes. We found it took 94.53 seconds, i.e. 26 times longer than before.

To repeat the test with procedures, start with Program 3.4:

```
 10 REM PROC speed test
 20
 30 TIME = 0
 40 FOR I% = 1 TO 5000
 50   PROCtest
 60   NEXT I%
 70 PRINT TIME / 100
 80
 90 END
100
110 DEF PROCtest
120 ENDPROC
```

Program 3.4

We found that this took 4.42 seconds but was no slower when we added the extra 190 lines of REMs.

Passing parameters to a procedure

So far, procedures have two advantages over subroutines: they can be called by name instead of by line number and they do not slow down as programs get longer. Being able to call a procedure by name saves us the bother of remembering where the procedure is and provides us with a hint as to what the procedure does, e.g.

```
PROCwait
PROCdraw_circle.
```

There is one disadvantage which arises from procedures not being callable by line number: you don't know where they are. With short programs this is not much of a problem but when programs are hundreds of lines long, finding a procedure can be difficult. One solution to this is to keep procedures in alphabetical order in your programs. Another is to keep procedures in some logical order, the choice of order depending on the nature of the program.

Procedures really come into their own when we use the fact that we can 'pass parameters' to them. That is, we can tell the procedure what value(s) one or more variables are to have in it. To do this we have to tell the procedure how many parameters to expect and we have to send it the correct number of parameters in the correct order. For example, in Program 3.2 we used PROCinput to input strings up to 6 characters long. We can now modify it so that we can tell it how long the next string is to be. The procedure is now defined as in Program 3.5.

```
130 DEF PROCinput(L)
140
150 REPEAT
160   INPUT A$
170   IF LEN(A$) > L THEN PRINT "Too long. Please
      re-enter."
180   UNTIL LEN(A$) <= L
190
200 ENDPROC
```

Program 3.5

We have made the length of the string a variable, L, in line 180. Then in line 130 we have changed DEF PROCinput to

```
DEF PROCinput(L).
```

This tells the program to expect a value for L whenever the procedure is called. If we want L to be 6, we write

```
PROCinput(6)
```

(Note that you can leave a space between the procedure name and the list of parameters e.g. `PROCinput (L)`.)

Program 3.6 shows Program 3.2 modified to call `PROCinput(L)` on the same two occasions as before with both input lengths set at 6 characters.

```
 10 REM Input check
 20
 30 PRINT "Enter name (max 6 characters)"
 40 PROCinput(6)
 50 name$ = A$
 60
 70 PRINT "Enter area code (max 6 characters)"
 80 PROCinput(6)
 90 area$ = A$
100
110 END
120
130 DEF PROCinput(L)
140
150 REPEAT
160   INPUT A$
170   IF LEN(A$) > 6 THEN PRINT "Too long. Please re-enter."
180   UNTIL LEN(A$) <= 6
190
200 ENDPROC
```

Program 3.6

`PROCinput(L)` can be passed any value of `L` when it is called. However, the procedure itself must be able to accept it, e.g. a negative number will cause problems in Program 3.6.

SAQ 2

Modify Program 3.6 in order to use the procedure to make the maximum length of name 10 characters and the maximum length of area code 3 characters.

When the parameter is a variable

In the examples so far, the parameters passed to the procedure were numbers (6, 10 and 3). Usually we find that the value that we wish to pass is a variable. Procedures cope with this in exactly the same way as with numbers: the variable to be passed is placed in brackets after the call, e.g.

```
PROCinput(W)
```

would call `PROCinput(L)` and give `L` the value held by `W`. Notice that to call the procedure we don't need to know the name of the variable to be

specified. We only need to know that a parameter needs to be passed. Suppose that in the input example the lengths of each input had been determined earlier in the program. They would then appear as variables to be passed to PROCinput(L) as in Program 3.7.

```
 10 REM Input check
 20
 30 name_len = 10 : REM Length of name
 40 code_len = 3 : REM Length of area code
 50
 60 PRINT "Enter name (max " ; name_len " characters)"
 70 PROCinput(name_len)
 80 name$ = A$
 90
100 PRINT "Enter area code (max " ; code_len " characters)"
110 PROCinput(code_len)
120 area$ = A$
130
140 END
150
160 DEF PROCinput(L)
170
180 REPEAT
190   INPUT A$
200   IF LEN(A$) > L THEN PRINT "Too long. Please re-enter."
210   UNTIL LEN(A$) <= L
220
230 ENDPROC
```

Program 3.7

K Type and run Program 3.7. Try changing the values of name_len and code_len in lines 30 and 40.

There is no limit to the number of parameters that we can pass to a procedure. The only conditions to fulfil are (a) that we must send as many parameters to the procedure as the procedure is expecting to receive and (b) we must send the parameters in the order they appear in the DEF PROC statement. The mixture of parameters may, of course, be a mixture of numeric and string variables. For example, PROCbirthday in Program 3.8 prints a birthday message to the person whose name is passed to it, repeated once for every year of the person's age.

```
 10 REM Passing parameters to a procedure
 20
 30 INPUT "Name " name$
 40 INPUT "Age " age
 50 PROCbirthday(name$, age)
```

```
 60
 70 END
 80
 90 DEF PROCbirthday(name$, age)
100
110 FOR I = 1 TO age
120   PRINT "Happy birthday, " name$
130   NEXT I
140
150 ENDPROC
```

Program 3.8

```
>RUN
Name Someone
Age 4
Happy birthday, Someone
Happy birthday, Someone
Happy birthday, Someone
Happy birthday, Someone
```

SAQ 3

Write a program in which the user enters a number of columns, '`cols`', a number of rows, '`rows`', and a string in `L$`. Send these variables to a procedure that will print the string '`cols`' times in each of '`rows`' rows.

Restrictions on parameters

You can use any valid expression as a parameter, e.g. the following would be valid provided all the variables have been assigned values:

```
PROCaction(A + B, COS(X * Y), "Hello" + STR$(Q), VAL(P$))
```

Local variables

The final general point about procedures concerns 'local variables'. In a procedure you may wish to re-use variable names which are already used elsewhere in the program. For example, in Program 3.1b, the variable '`I`' might already be in use in the host program into which we put `PROCdraw`. By adding the line `LOCAL I` before we use `I` inside the procedure, we ensure that the `I` in the procedure will be a distinct variable from the `I` used elsewhere in the program. Program 3.1c demonstrates this.

```
10 REM Local variables
20
30 REM I in this loop is a different
40 REM variable from I in PROCdraw
50 FOR I = 1 TO 10
60 PROCdraw
70 NEXT I
```

```
 80
 90 END
100
110 DEF PROCdraw
120 LOCAL I
130 FOR I = 1 TO 5
140   PRINT "*" ;
150 NEXT
160 PRINT
170 ENDPROC
```

Program 3.1c

Notice that any variables listed in the procedure definition line are automatically local to that procedure and need not be listed in the 'LOCAL' line. You must be careful, therefore, not to expect one of these local variables to return a changed parameter value, e.g.

```
100 PROCadd(A)
110 PRINT A
    .
    .
200 DEF PROCadd(A)
210 A = A + 1
220 ENDPROC
```

will *not* result in line 110 printing A incremented by 1 since the A in PROCadd is local to that procedure.

Where to put procedures

Procedures should always appear below the END statement of your program.

Entering and leaving procedures

Procedures must only be entered by a PROC statement. If you use GOTO (whether to the beginning or the inside of a procedure) the computer will lose track of its route through your program as soon as it reaches ENDPROC. Equally you must not jump out of a PROC using GOTO.

ON ERROR and procedures

Procedures follow the same rules about ON ERROR as GOSUBs. These were discussed under the heading 'Three warnings' in section 2.5.

Style and structure

Procedures enable you to write clearer, more understandable programs. We shall say more about this in Unit 5. However, to get the best out of procedures, you must make them as general as possible – avoid limiting them by the circumstances of the problem that you are currently trying to solve. Program 3.9 is an example of a generalised procedure. It allows you to print any text string over a width chosen by you without

the words being broken up. You can drop this procedure into any program since all variables in it are local: no adjustments are needed to meet conditions in the host program. Notice that, to keep it general, it doesn't even execute a carriage return on exit since that will not always be needed.

```
 10 REM A procedure for neat text printing
 20
 30 F$ = "This is a piece of text to be printed without
    wordbreaks.  This is a piece of text to be printed without
    wordbreaks. This is a  piece of  text to be printed without
    wordbreaks."
 40 length = 18
 50
 60 PROCword_print(F$, length)
 70
 80 END
 90
100 DEF PROCword_print(P$, L)
110 LOCAL word$, I, done
120
130 done = FALSE
140 REPEAT
150   I = INSTR(P$, " ")
160   word$ = LEFT$(P$, I)
170   P$ = RIGHT$(P$, LEN P$ - I)
180   IF P$ = "" THEN done = TRUE
190   IF I = 0 THEN  word$ = P$ : done = TRUE
200   IF LEN word$ > L - COUNT THEN PRINT
210   PRINT word$ ;
220   UNTIL done
230
240 ENDPROC
```

```
>RUN
This is a piece
of text to be
printed without
wordbreaks. This
is a piece of
text to be
printed without
wordbreaks. This
is a piece of
text to be
printed without
wordbreaks.>
```

Program 3.9

(Don't worry about INSTR and COUNT if you have not met them – it is the structure of the procedure that we want you to look at.)

[K] Try Program 3.9 with some different line lengths.

SAQ 4

Write a procedure to check whether a string which should contain only letters either (a) is longer than a maximum length sent to it, or (b) contains numeric characters. (Assume that the invalid characters will be one of 1,2,3, . . . 9).

Procedures versus subroutines

Procedures can do everything that subroutines can do and provide the valuable facilities of parameter passing and local variables. For example, you could not replace lines 110 to 170 of Program 3.1c with a subroutine such as

```
110 REM Draw subroutine
130 FOR I = 1 TO 5
140   PRINT "*";
150   NEXT I
160 PRINT
170 RETURN
```

since the variable I is already being used elsewhere in the program. *Every* variable in a subroutine has to be a global variable. Equally, you cannot pass parameters to a subroutine, so the procedure in Program 3.9 could not be replaced with a subroutine using the same variables. It would have to use F$ and 'length'. The importance of this to modular design is more fully developed in Unit 5.

3.2 User-defined functions

User-defined functions are very similar to procedures but are used to return values to the statement which called them. A defined function can be used to do anything done by a procedure although most programmers use procedures unless the 'value returning' aspect of the defined function is needed. For example, you might use a function to:

calculate an average
work out a square root
find the cosine of an angle
work out an area
work out your income tax for one month.

(Notice that the second and third items are functions already in BBC BASIC, i.e. SQR and COS are functions. In this part of the book we are concerned with user-defined functions – ones which you create.)

We particularly use defined functions when the value returned will be

of special significance in the program. We shall explain this in more detail once we have shown how defined functions work.

Defining a function

A user-defined function has three parts to it:

1. The line which starts the definition shown here with two parameters being passed to it — `DEF FNname(p1, p2)`
2. The body of the definition.
3. The end line which returns the calculated value. — `= value`

Notice the form of the end line (`= value`). This means that the number 'value' will be sent back by the function to the statement which called it.

The function is an operator which affects a variable. So to use `FNname` we can write

```
T = FNname(p1, p2)
```

or

```
PRINT FNname (p1, p2)
```

Example 1

In a simple tax system, tax is levied at two rates: the lower rate of 30% on income between £5001 and £10,000 and the higher rate of 40% on income above that. Income of £5000 or below is not taxed. We will write a user-defined function which is sent a person's income and returns the tax he has to pay.

Solution

We shall call the function `FNtax`. It needs one parameter which is the person's income. Since all it calculates is tax due, only one parameter is returned so a user-defined function will do the job.

```
10 REM User-defined function demonstration
20
30 INPUT "Enter taxable income " income
40
50 tax = FNtax(income)
60
70 PRINT "Tax payable is " ; tax
80
90 END
```

```
100
110 DEF FNtax(income)
120
130 IF income < 5001 THEN tax = 0
140 IF income >= 5001 THEN tax = (income - 5000) * 0.3
150 IF income >= 10001 THEN tax = tax + (income - 10000)
    * 0.1
160 = tax

>RUN
Enter taxable income 4500
Tax payable is 0

>RUN
Enter taxable income 7500
Tax payable is 750

>RUN
Enter taxable income 12500
Tax payable is 2500
```

Program 3.10

SAQ 5

Write a user-defined function which is sent a number as a string and adds up the digits in the number, returning the total as the value of the function.

Local variables in defined functions

Local variables can be declared in defined functions in the same way as in procedures. The variables `I%` and `sum%` in Program 3.11 are declared as local to the function. The function itself forms the 'checksum' for a block of memory. The checksum is the sum of all the values in the memory locations and is part of the error-checking system used in saving and loading programs.

```
10 REM Checksum
20
30 INPUT "Enter start address in decimal " start
40 INPUT "Enter finish address in decimal " finish
50
60 PRINT "Checksum = "; FNcheck_sum(start, finish)
70
80 END
90
100 DEF FNcheck_sum(start, finish)
110 LOCAL I%, sum%
120 sum% = 0
```

```
130 FOR I% = start TO finish
140   sum% = sum% + ?I%
150   NEXT
160 = sum%

>RUN
Enter start address in decimal 6400
Enter finish address in decimal 6696
Checksum = 23197
```

Program 3.11

K Type and run Program 3.11.

One-line definitions

Where the definition of a function is very short, it may be convenient to write it in one line. For example, the following defines a one-line function which adds the squares of the two parameters sent to it:

```
DEF FNsquares(A, B) = A ^ 2 + B ^ 2
```

You should now be able to see how useful defined functions are when we want to use the returned value in a program. Here are some examples of the kind of uses that you can make of defined functions:

```
PRINT FNlargest(A, B)
IF FNvalid(N$) THEN PROCfile
IF FNxyz(A$) AND FNabc(A$) THEN ...
```

Where to put function definitions

As with procedure definitions, function definitions must be placed at the end of the program where they cannot be executed without being called from the main part of the program. One-line functions are an exception to this rule. You may place one-line function definitions anywhere in a program and allow the program to fall through them. However, your programs will be easier to follow if *all* definitions (procedures and functions) are grouped together after the `END` statement.

SAQ 6

Write a one-line user-defined function to calculate the sum of two numbers divided by their product.

3.3 TIME, MOD and DIV

These are not user-defined functions but since we need them for the next few units we shall deal with them now.

TIME

The BBC Micro has an internal clock which controls all of its operations. It produces two million 'ticks' per second. These ticks are

electronically scaled down to 100 per second for the use by programmers. These scaled-down ticks are then counted by TIME.

Try Program 3.12 to see how TIME is recorded.

```
10 REM TIME demonstration
20
30 FOR I = 1 TO 10
40   FOR J = 1 TO 10
50     PRINT ; TIME " " ;
60     NEXT J
70   PRINT
80   NEXT I
```

```
>RUN
701613 701615 701616 701618 701620 701622 701623 701625 701627 701629
701631 701632 701634 701637 701643 701648 701654 701660 701666 701672
701678 701684 701690 701696 701702 701707 701713 701719 701725 701731
701738 701743 701749 701755 701761 701767 701772 701778 701784 701790
701797 701802 701808 701814 701820 701826 701831 701837 701843 701849
701856 701861 701867 701873 701879 701885 701890 701896 701902 701908
701915 701920 701926 701932 701938 701944 701950 701955 701961 701967
701974 701979 701985 701991 701997 702003 702009 702014 702020 702026
702033 702039 702044 702050 702056 702062 702068 702073 702079 702085
702092 702098 702103 702109 702115 702121 702127 702133 702138 702144
```

Program 3.12

This shows that at the first execution of line 50 TIME was 701613 hundreths of a second and at the last execution it was 702144 hundreths of a second. So the run took 5.31 seconds. (The digits in your answer will depend upon how long your computer has been switched on. Run time can vary according to how the computer is being used. On our computer the slow run time of 5.31 seconds resulted from the program being held up by the printer. Without the printer, the run time was 1.25 seconds).

SAQ 7

(a) Rewrite Program 3.12 to print out the computer's time in seconds instead of hundredths of a second.

(b) Make your new program display the time when the run started and the time when it finished.

SAQ 8

Write a procedure which will cause a delay for whatever number of seconds that is sent to the procedure.

The procedure which you have just developed is an extremely useful

one. It can be used to present instructions to a user for a fixed period or to present a sequence of information panels. Suppose, for example, that you found that readers needed 1 second per two lines of text. To display a panel of 6 lines you would print the panel and then call `PROCwait(3)`. To display 10 lines you would call `PROCwait(5)`.

The computer's `TIME` is convenient for measuring short periods but awkward for longer periods. It is therefore useful to be able to convert `TIME` to count in seconds, minutes and hours. To do this we need to introduce two new statements, `MOD` and `DIV`.

Re-setting TIME

If you want to set `TIME` to a new starting value then you can do this by treating `TIME` as a variable. For example,

```
TIME = 0
```

will re-set `TIME` to zero. Immediately after this statement has been executed, `TIME` starts to increment again at the rate of 100 per second.

MOD

`MOD` stands for modulo and arises in a special form of division where only the remainder is recorded. To find

```
5 MOD 2
```

divide 5 by 2 and take the remainder, i.e. 1, so

```
5 MOD 2 = 1.
```

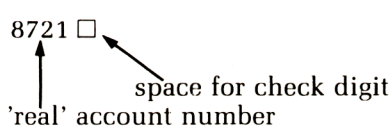
SAQ 9

(a) Find `1 MOD 2, 2 MOD 2, 3 MOD 2, 4 MOD 2, 5 MOD 2, 6 MOD 2.`
(b) Find `1 MOD 5, 2 MOD 5, 3 MOD 5, 4 MOD 5, 5 MOD 5, 6 MOD 5.`

From this you can see that `MOD 8`, say, will always produce one of 0, 1, 2 ... 7 and numbers `MOD 12`, say, will produce one of 0, 1, 2 ... 11.

Check digits

When a record (i.e. a collection of information on something or someone held in a standard form) in a computer is identified solely by a number (e.g. an account number), the number should contain a built-in check to guard against incorrect typing. The simplest systems place a '*check digit*' at the end of the account number, e.g.



One such system finds the check digit by applying the following rule:

```
(n * (d + 1) + C) MOD 11 = 0
```

where n = actual digit, d = digit position and C is the check digit. Because the check digit is the result of a modulo 11 calculation it can be any number between 0 and 10. If it is 10 then it is written as 'X' – something you may well have seen as the end of International Standard Book Numbers on books e.g.

0 85312 423 X

SAQ 10

Write a user-defined function which will check whether numbers sent to it are correct according to this rule. If correct, the function should send back the value -1 (`TRUE`); if incorrect it should send back the value 0 (`FALSE`). Test your function on 6213X, 6231X, 2613X and 2631X to find out which is the one which has been correctly typed. What errors are present in the incorrect ones?

DIV

`DIV` is complementary to `MOD`, i.e. to find

```
5 DIV 2
```

perform the division and record the number of complete divisions, ignoring the remainder, so

```
5 DIV 2 = 2
```

since 2 goes into 5, 2 times, remainder 1.

SAQ 11

What is `10 DIV 3, 1 DIV 5, 12 DIV 12, 33 DIV 5, 50 DIV 60`?

We can now use `MOD` and `DIV` to convert `TIME` into seconds, minutes and hours.

```
TIME DIV 100
```

gives the number of whole seconds since the computer was switched on or since the value of `TIME` was altered.

```
TIME DIV 6000
```

gives the number of whole minutes since the computer was switched on or since the value of `TIME` was altered.

```
TIME DIV 360000
```

gives the number of whole hours since the computer was switched on or since the value of `TIME` was altered.

Try running Program 3.13 to see what the current '*time*' is on your computer.

```
 10 REM A crude clock
 20
 30 REPEAT
 40
 50   SEC = TIME DIV 100
 60   MIN = TIME DIV 6000
 70   HR = TIME DIV 360000
 80
 90   PRINT ; HR, ; MIN, ; SEC
100
110   UNTIL FALSE
```

Program 3.13

K Type and run Program 3.13.

This program will count hours, minutes and seconds. If you want to see the time as on a clock with seconds and minutes between 0 and 59 and hours between 0 and 11 then you need to add `MOD 60`, `MOD 60` and `MOD 12` respectively.

```
SEC = (TIME DIV 100) MOD 60
MIN = (TIME DIV 6000) MOD 60
HR = (TIME DIV 360000) MOD 12
```

SAQ 12

Write a program which prints a digital clock at the centre of the screen like this:

```
9  : 35 : 23
```

where the numbers are in the order hours, minutes, seconds.

3.4 TAB(X, Y)

`TAB(X,Y)` is used in the same way as `TAB(X)` but it allows us to specify the vertical and horizontal positions of the cursor. The value of Y depends on the mode used. In modes which gives 32 lines, Y will be 0 - 31. In modes which give 25 lines, Y will be 0 - 24.

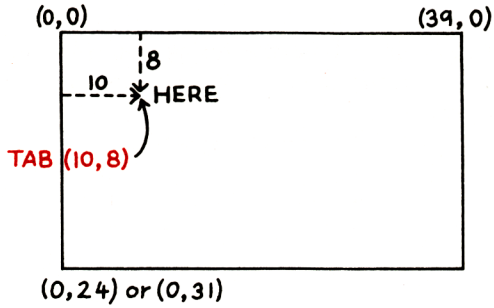


Figure 3.1 Tabbing to 10 characters in on the 9th line

The statement `PRINT TAB(10, 8) "HERE"` will print `HERE` 9 lines down and 10 characters in from the left hand side. This has a number of uses, the most obvious of which is to display text on the screen without having to use empty `PRINT` statements to move the cursor down the screen.

Example 2
Write a program to display the following:

```
DEMONSTRATION
     OF
  TAB(X, Y)
```

on lines 7, 12 and 19.

Solution

```
 10 REM TAB(X, Y) demonstration
 20
 30 MODE 7
 40
 50 PROCcentre(7, "DEMONSTRATION")
 60 PROCcentre(12, "OF")
 70 PROCcentre(19, "TAB(X,Y)")
 80
 90 END
100
110 DEF PROCcentre(line, text$)
120 PRINT TAB((40 - LEN text$) DIV 2, line) text$
130 ENDPROC
```

Program 3.14

In line 120 we have used `DIV 2` where we could have used `/2`. `DIV` is faster than `/` and, since we only want the whole number part of the result, `DIV` is more appropriate than `/`.

K Type and run Program 3.14.

SAQ 13

Write a procedure to print a box of asterisks as in Figure 3.2 where the procedure is sent the coordinates of the centre of the box and the width and height of the box.

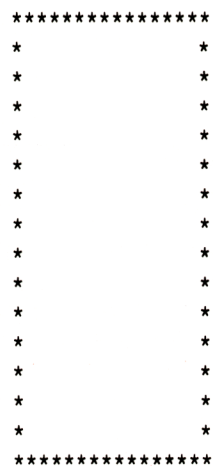


Figure 3.2

SAQ 14

BBC BASIC has a facility for printing double height characters in Mode 7. To print, say, `I'M BIGGER THAN YOU` in double height characters you write:

```
150 PRINT CHR$(141) "I'M BIGGER THAN YOU"
160 PRINT CHR$(141) "I'M BIGGER THAN YOU"
```

The two lines are identical and must be written with no other print statements between them. Write a procedure which will print a string in double-height characters at a specified position i.e. pass to the procedure the string and its horizontal and vertical `TAB` positions.

Screen input

We shall now apply `TAB(X,Y)` to the problem of producing a screen input display system. If you have seen computers in commercial or industrial environments, you will be aware of the difference between the screen input and the typical microcomputer system. First, let us demonstrate the problem. Suppose you have a program which requires many names, addresses and telephone numbers to be entered. The screen prompts might be something like:

```
>RUN
Next record
Name    ?MR JONES
Address ?MILL LANE
Tel No. ?34576
```

```
Next record
Name    ?MRS SMITH
Address ?45 END OF THE WORLD
Tel No. ?98703

Next record
Name    ?MX
Address ?
```

Apart from the mess on the screen, this example demonstrates how impossible this system is to use. The user made a mistake in the last name and instead of pressing Delete, pressed Return. At that point, it became impossible to correct the name.

To overcome the problem, it is better to offer the user a clear screen with all the field titles and then to ask him to enter the fields in any order he likes. The screen might come up showing:

```
CUSTOMER ADDRESS FILE
Name
Address
Address
Address
Tel No.
```

To develop the full system required to run this type of input, we need to use `GET`, which we introduce later in this course. So, at this stage, we will develop a system which produces something very near to the full screen input system.

Example 3
Write a program to display:

```
CUSTOMER'S ADDRESS FILE
1 Name
2 Address1
3 Address2
4 Address3
5 Tel No.
6 Entry complete
Enter number required:
```

The maximum length of each field (i.e. space in the record for information) should be: name, 15; address (each line), 20; Tel No., 10.

We are going to develop this as two procedures which you can use in other programs with minimal modification. The program uses three arrays:

`T$( )` which holds the field titles.

`L( )` which holds the maximum lengths permitted for each field.

`F$( )` which holds the entries for the material at present on the screen.

The solution is in two parts: `PROCscreen` to set up the screen and `PROCinput` for the user to enter his information.

Set up the screen

This clears the screen and prints the field titles using `TAB(X, Y)`. For convenience we have loaded the maximum field lengths inside this procedure despite the fact that the procedure doesn't use them. This may seem odd, but it's convenient to read all the screen data for one input operation at the same time (lines 160 to 200). The field names are then put into `T$( )` and are used by the procedure; the field lengths are stored in `L( )` ready for the following procedure.

The procedure is shown in Program 3.15.

```
120 DEF PROCscreen(title$)
130 LOCAL I
140 CLS
150 REM Field lengths and titles
160 RESTORE 310
170 READ fields
180 FOR I = 1 TO fields + 1
190   READ L(I), T$(I)
200   NEXT I
210
220 PRINT TAB((width - LEN title$) DIV 2, 2) title$
230
240 FOR I = 1 TO fields + 1
250   PRINT TAB(0, 5 + I) ; I TAB(3, 5 + I) T$(I)
260   NEXT I
270
280 ENDPROC
290
300 REM No of fields
310 DATA 5
320 REM Field lengths and titles
330 DATA 15, Name, 20, Address1, 20, Address2
340 DATA 20, Address3, 10, Tel No., 0, Entry complete
```

Program 3.15

Next we set up the input procedure. The flow chart for this is shown in Figure 3.3.

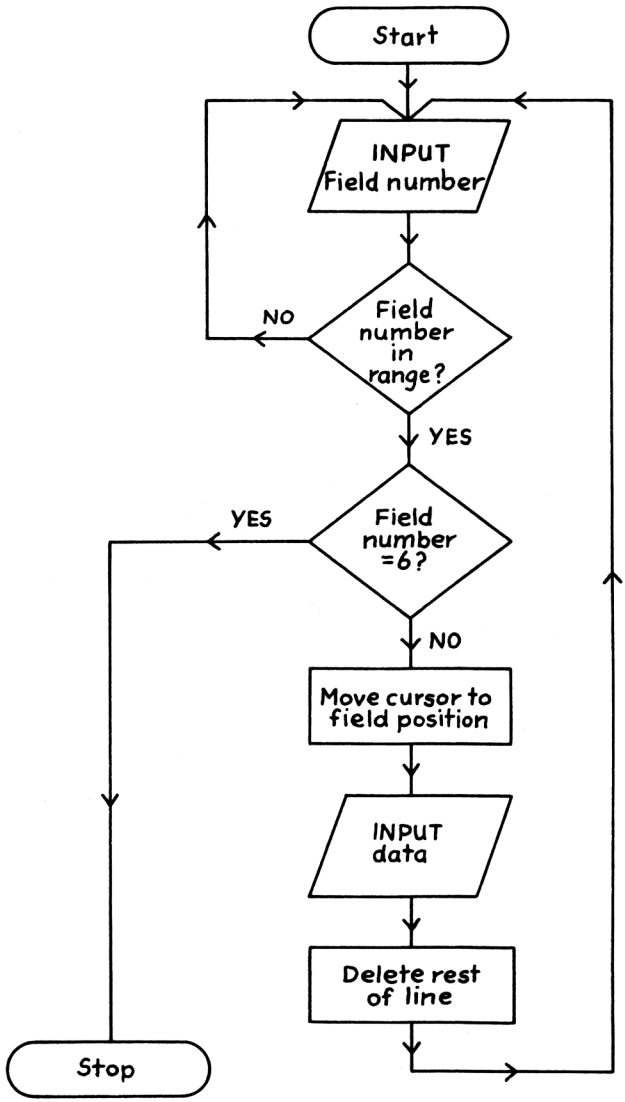


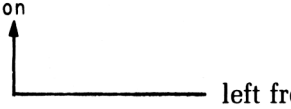
Figure 3.3 Flowchart for input procedure

The only part that needs a special explanation is '*delete rest of line*'. Suppose the user had entered:

```
1. Name  Mr A J Johnson
```

and then realised the name was wrong. He selects field 1 again and enters the correct name (Mr A J Jones) and the screen shows

```
1. Name  Mr A J Jones on
```



left from previous entry.

As soon as he presses return, the '*on*' will be deleted. This is done by lines 540 - 560.

```
360 DEF PROCinput(fields)
370
380 REPEAT
390   REPEAT
400
410     REM Choose field number
420     PRINT TAB(0, 20) "Enter number required."
430     INPUT F
440     PRINT TAB(1, 21) SPC(39)
450     UNTIL F > 0 AND F <= fields + 1
460
470   IF F = fields + 1 THEN CLS : ENDPROC
480
490   REM Move cursor to field F and enter data for
      that field
500   PRINT TAB(13 + L(F), 5 + F) "." TAB(12, 5 + F) ;
510   INPUT "" F$(F)
520
530   REM Blank out field F to right of cursor
540   FOR I = LEN(F$(F)) + 1  TO 28
550     PRINT TAB(11 + I, 5 + F) " " ;
560     NEXT I
570
580   UNTIL F = fields + 1
590 CLS
600
610 ENDPROC
```

Program 3.16

This procedure can be used exactly as it stands provided (a) the number of fields isn't too large for the screen and (b) no field runs over the end of a line. We have set the procedure up for a 40-character line display which limits the maximum field length to 27 (see line 540). If you were to run it in Mode 0, the maximum field length would be 67 and the 28 in line 540 should be changed to 68.

[K] To test these two procedures, set up the following:

```
10 DIM T$(6), L(6), F$(6)
15 fields = 5
20 PROCscreen("CUSTOMERS' ADDRESS FILE")
30 PROCinput(fields)
40 REM Check that it worked
50 FOR I = 1 TO fields
60   PRINT F$(I)
70   NEXT I
80 END
```

SAQ 15

If a user entered a name and address and then exited by typing 6, where would the data be? How could the procedures be used for the next address?

Assignment C

1. Write a procedure to clear the screen and print (in Mode 7) a string sent to it in double height characters in the middle of the centre line of the screen. (Assume the string will not exceed one line in length).

2. Write a procedure to sort `A$( )` into alphabetical order where `A$( )` is a global variable and contains *N* items. Make as many as possible of the variables in your procedure local. Reminder: the standard bubble sort (when a flag is not being used) for an array of *N* items is:

```
100 FOR I% = 1 TO N - 1
110   FOR J% = I% + 1 TO N
120     IF A$(J%) < A$(I%) THEN T$ = A$(J%) : A$(J%) = A$(I%) :
        A$(I%) = T$
130     NEXT J%
140   NEXT I%
```

3. Write a procedure to reverse the characters in a string, i.e. if sent the string `ABCD` it returns `DCBA`. The procedure should be able to cope with strings of any length.

Objectives of Unit 3

At the end of this unit you should be able to:

- ☐ Replace a subroutine with a procedure in a program.
- ☐ Write procedures which require several parameters to be passed to them.
- ☐ Call procedures which require several parameters to be passed to them.
- ☐ Use local variables inside a procedure.

- ☐ Write one line and multi-line user-defined functions.
- ☐ Use local variables inside a user-defined function.
- ☐ Use `TIME` in programs.
- ☐ Use `MOD` and `DIV` for modular arithmetic calculations.
- ☐ Use `TAB(X, Y)` to print characters at any point on the screen.

Answers to SAQs

SAQ 1

```
 10 REM Input check
 20
 30 PRINT "Enter name (maximum 6 characters)"
 40 PROCinput
 50 name$ = A$
 60
 70 PRINT "Enter area code (maximum 6 characters)"
 80 PROCinput
 90 area$ = A$
100
110 END
120
130 DEF PROCinput
140
150 REPEAT
160   INPUT A$
170   IF LEN A$ > 6 THEN PRINT "Too long. Please re-enter."
180   UNTIL LEN(A$) <= 6
190
200 ENDPROC
```

Program 3.17

SAQ 2

```
 10 REM Input check
 20
 30 PRINT "Enter name ";
 40 PROCinput(10)
 50 name$ = A$
 60
 70 PRINT "Enter area code ";
 80 PROCinput(3)
 90 area$ = A$
100
110 END
120
```

```
130 DEF PROCinput(L)
140
145 PRINT "(maximum "; L " characters)."
150 REPEAT
160   INPUT A$
170   IF LEN A$ > L THEN PRINT "Too long. Please re-enter."
180   UNTIL LEN A$ <= L
190
200 ENDPROC
```

Program 3.18

```
>RUN
Enter name (maximum 10 characters).
?JONATHAN SMITH
Too long. Please re-enter.
?J SMITH
Enter area code (maximum 3 characters).
?BERKSHIRE
Too long. Please re-enter.
?BKS
```

SAQ 3

```
 10 REM Procedure needing 3 parameters
 20
 30 INPUT "Number of columns " cols
 40 INPUT "Number of rows    " rows
 50 INPUT "A string of characters " L$
 60
 70 PROCprint_pattern(cols, rows, L$)
 80
 90 END
100
110 DEF PROCprint_pattern(A, B, S$)
120 LOCAL I, J
130 CLS
140
150 FOR I = 1 TO B
160   FOR J = 1 TO A
170     PRINT S$ ; " " ;
180     NEXT J
190   PRINT
200   NEXT I
210
220 ENDPROC
```

Program 3.19

```
>RUN
Number of columns 3
Number of rows    4
A string of characters WHAT?
WHAT? WHAT? WHAT?
WHAT? WHAT? WHAT?
WHAT? WHAT? WHAT?
WHAT? WHAT? WHAT?
```

K Type and run Program 3.19.

SAQ 4

```
 10 REM String checking for non-valid characters
 20
 30 INPUT "Maximum length " maxlen
 40 INPUT "Enter string " A$
 50
 60 PROCstring_check(A$, maxlen)
 70 END
 80
 90 DEF PROCstring_check(A$, maxlen)
100 LOCAL I
110
120 FOR I = 1 TO LEN A$
130    IF  VAL(MID$(A$, I,  1)) <>  0 THEN  PRINT "String
       contains invalid character at position " ; I
140   NEXT I
150
160 IF LEN A$ > maxlen THEN PRINT "String  is too long -
    maximum length is " ; maxlen " characters."
170 ENDPROC

>RUN
Maximum length 5
Enter string F4HG9NB
String contains invalid character at position 2
String contains invalid character at position 5
String is too long - maximum length is 5 characters.
```

Program 3.20

This is not a very good check since it allows '0', an invalid character, through. More effective routines will be developed in this course as we introduce other BASIC commands. In particular, `INSTR` will provide a more effective way of expressing `VAL(MID$(A$, I, 1))`.

SAQ 5

```
 10 REM Character count
 20
 30 INPUT "Enter code number " number$
 40 PRINT "Total value of numbers in code = ";
    FNcode_total(number$)
 50
 60 END
 70
 80 DEF FNcode_total(X$)
 90 LOCAL I, T
100 T = 0
110
120 FOR I = 1 TO LEN X$
130   T = T + VAL(MID$(X$, I, 1))
140   NEXT I
150
160 = T

>RUN
Enter code number 123ABC456
Total value of numbers in code = 21
```

Program 3.21

SAQ 6

```
100 DEF FNspecial(A, B) = (A + B) / (A * B)
```

SAQ 7

(a)
```
10 REM TIME demonstration
20
30 FOR I = 1 TO 10
40   FOR J = 1 TO 10
50     PRINT ; TIME / 100 " " ;
60     NEXT J
70   PRINT
80   NEXT I
```

Program 3.22

K Type and run Program 3.22.

(b)
```
 10 REM TIME demonstration
 20
 30 PRINT "Start at " ; TIME / 100 '
 40 FOR I = 1 TO 10
 50   FOR J = 1 TO 10
 60     PRINT ; TIME / 100 " " ;
```

```
 70      NEXT J
 80    PRINT
 90    NEXT I
100 PRINT "End at " ; TIME / 100
```

Program 3.23

K Type and run Program 3.23.

SAQ 8

```
1000 DEF PROCwait(X)
1010 finish = TIME + X * 100
1020 REPEAT
1030   UNTIL TIME >= finish
1040 ENDPROC
```

Program 3.24

K Type and run Program 3.24.

Notice the use of '>=' in line 1030. If you use '=' the condition may never be met and the procedure will repeat the loop forever.

SAQ 9

(a) 1 MOD 2 = 1
2 MOD 2 = 0 (2 divides exactly by 2 so the remainder is 0)
3 MOD 2 = 1
4 MOD 2 = 0 (4 divides exactly by 2)
5 MOD 2 = 1
6 MOD 2 = 0 (6 divides exactly by 2)

(b) 1 MOD 5 = 1
2 MOD 5 = 2
3 MOD 5 = 3
4 MOD 5 = 4
5 MOD 5 = 0 (5 divides exactly by 5)
6 MOD 5 = 1
etc.

SAQ 10

```
 10 REM Checking numbers modulus 11
 20
 30 REPEAT
 40   REPEAT
 50     INPUT "Number to check " number$
 60     ok = FNnumber_check(number$)
 70     IF NOT ok THEN PRINT "Invalid number.
        Please re-enter."
 80     UNTIL ok
 90   PRINT "Number OK. "
100   UNTIL FALSE
```

```
110
120 END
130
140 DEF FNnumber_check(X$)
150 LOCAL I, T, L, E
160 T = 0
170 L = LEN X$
175 dud_char = FALSE
180
190 FOR I = 1 TO L
200   T$ =MID$(X$, I, 1)
205   IF T$ <> "X" AND T$ <> "0" AND VAL T$ = 0 THEN
      dud_char = TRUE
210   IF T$ = "X"  THEN number =10 ELSE number = VAL T$
220   T = T + number * (L - I + 1)
230   NEXT I
240
250 IF dud_char OR T MOD 11 THEN = FALSE ELSE = TRUE
```

```
>RUN
Number to check 6213X
Invalid number. Please re-enter.
Number to check 6231X
Invalid number. Please re-enter.
Number to check 2613X
Invalid number. Please re-enter.
Number to check 2631X
Number OK.
```

Program 3.25

The errors which we made all involved accidental transposing of adjacent digits in the correct number. This is a common typing error which is usually trapped by a modulo 11 check digit.

SAQ 11

```
 10 DIV 3  =  3
  1 DIV 5  =  0
 12 DIV12  =  1
 33 DIV 5  =  6
 56 DIV60  =  0
```

SAQ 12

Your solution should be something like:

```
10 REPEAT
20   MODE 7
30   SEC = (TIME DIV 100) MOD 60
40   MIN = (TIME DIV 6000) MOD 60
50   HR = (TIME DIV 360000) MOD 12
60   PRINT '''''''''''''
70   PRINT TAB(12) ; HR ; TAB(14) ":" ; MIN ;
     TAB(17) ":" ; SEC
80   UNTIL FALSE
```

Program 3.26

This has two problems: (a) the need to clear the screen and reprint the clock on each pass round the loop. This we will cure shortly using `TAB(X, Y)`; (b) the reprinting happens so frequently that the screen flickers very unpleasantly. The cure for this is straightforward; we put a pause at the end of the loop:

```
75 REPEAT
76 UNTIL TIME MOD 100 = 0
```

This extra loop repeats itself until the end of the current second.

K Type and run Program 3.26 with lines 75 and 76.

SAQ 13

```
 10 REM Print a box with stars
 20
 30 REM Test routine
 40 INPUT "Enter width of box " W
 50 INPUT "Enter height of box " H
 60
 70 CLS
 80 REM Centre on (19, 11)
 90 PROCbox(19, 11, H, W)
100
110 END
120
130 DEF PROCbox(X, Y, H, W)
140 w = W DIV 2
150 h = H DIV 2
160
170 PRINT TAB(X - w, Y - h) STRING$(W, "*")
180
190 FOR I = 1 TO H - 2
200   PRINT TAB(X  - w, Y - h + I) "*"
210   PRINT TAB(X + w + W MOD 2 - 1, Y - h + I) "*"
```

```
220   NEXT I
230
240 PRINT TAB(X - w, Y - h + I) STRING$(W, "*")
250
260 ENDPROC
```

Program 3.27

K Type and run Program 3.27.

SAQ 14

```
 10 REM Double height characters
 20
 30 MODE 7
 40
 50 PRINT TAB(0, 10) ;
 60 INPUT "Word to be printed? " word$
 70 INPUT "Horizontal position? " in
 80 INPUT "Vertical position? " down
 90
100 PROCdouble(word$, in, down)
110
120 END
130
140 DEF PROCdouble(A$, X, Y)
150 CLS
160
170 PRINT TAB(X, Y) CHR$141 A$
180 PRINT TAB(X) CHR$141 A$
190
200 ENDPROC
```

Program 3.28

K Type and run Program 3.28.

SAQ 15

On exit from `PROCinput(N)`, the data is in the array `F$( )`. If the procedure is to be used for another address, the data in `F$( )` must be transferred to other variables in the program. `F$( )` is used over and over again for each set of data which is entered.

UNIT 4

Graphics

4.1 What are graphics?

4.2 Graphics coordinates

4.3 DRAW

4.4 MOVE

4.5 PLOT 85

4.6 Colour selection 1

4.7 Colour selection 2

4.8 Text and graphics

4.9 Text and graphics windows

4.10 PLOT n, X, Y

Assignment D

Objectives of Unit 4

Answers to SAQs

4.1 What are graphics?

So far, everything that you have done has resulted in some kind of text (letters, numbers and special symbols such as *) appearing on the screen. Sometimes we have run these together e.g. **** or ---- to create a graphical image out of text, but it is still really text. Computer graphics are anything which appear on the screen *except* text. Try this program for some idea of what graphics can consist of:

```
 10 REM Graphics demonstration
 20
 30 MODE 5
 40 VDU 28, 3, 31, 16, 28
 50 VDU 29, 600 ; 500 ;
 60 GCOL 0,2
 70 MOVE - 400, - 300
 80 PLOT 22, - 400, 300
 90 PLOT 22, 400, 300
100 PLOT 22, 400, - 300
110 PLOT 22, - 400, - 300
120
130 REPEAT
140   INPUT "KEY 9 TO START " R
150   UNTIL R = 9
160
170 CLS
180 VDU 23;8202;0;0;0;
190 MOVE 0, 0
200
210 sector = 2 * PI / 15
230   FOR J = 0 TO .5 STEP .01
240     FOR I = 1 TO 15
250       GCOL 0, RND(4) - 1
260       MOVE 0, 0
270       angle = J + I * sector
280       DRAW 280 * COS(angle), 280 * SIN(angle)
290       NEXT I
300     NEXT J
310
320   PRINT "THAT'S ALL."
```

Program 4.1

K Type and run Program 4.1. Ignore what you don't understand and concentrate on seeing what it does.

You can see from this that graphics includes lines, dotted lines, colours and shapes. This unit will cover most of the basic graphics techniques on the BBC Micro and will apply them to useful projects, including drawing histograms and pie charts in colour.

4.2 Graphics coordinates

To draw something on the screen you must know where you want it to appear so that you can tell the computer to put it there. You have already met this idea in `TAB(X,Y)` which prints text at the position `(X,Y)` relative to an origin (the point from which all the measurements are made) at the *top* left-hand corner of the screen. BBC graphics co-ordinates are measured relative to an origin at the *bottom* left-hand corner and the graphics screen is 1280 points wide (points 0 - 1279) and 1024 points high (points 0 - 1023) – unless you have re-set the origin. (Even after re-setting the origin, the size of the graphics area is still 1280 points by 1024 points).

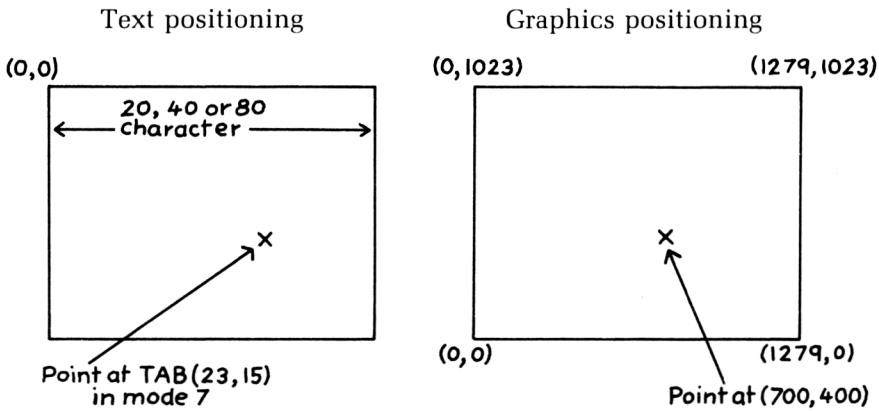


Figure 4.1 Text and graphics coordinates

If you are familiar with hex numbers, then you may find it useful to note that the graphics screen is 0 - &4FF by 0 - &3FF i.e. it is &500 points wide and &400 points high. Each character is &20 wide. (In standard Modes only). We have not used hex in this course, but once you know hex, it is easier to use hex numbers in BBC graphics since the arithmetic fits the screen better.

Graphics modes: some preliminaries

We shall go into more detail on the various modes in section 4.6 so at this stage we will simply state the minimum that you need to know to get the graphics to work. Only certain modes will show graphics – the rest show text only:

	Model A	Model B
Graphics modes	4,5	0,1,2,4,5
Text only modes	6,7	3,6,7

We shall mostly work in Mode 5 for graphics, and occasionally in Mode 4. To do this you start your program with the appropriate mode

number. Then you need a statement to select the graphics colour you want.

To get you started we shall do all the graphics in Mode 5 in yellow until we deal with how to make colour changes. We shall start with Program 4.2 which selects Mode 5 in line 30 and yellow in line 40.

```
10 REM Triangle
20
30 MODE 5
40 GCOL 0, 2
50 MOVE 200, 300
60 DRAW 500, 200
70 DRAW 800, 900
80 DRAW 200, 300
```

Program 4.2

[K] Type and run Program 4.2. Run the program, then type `LIST` to get it back on the screen. You will notice two things, one useful and one awkward. The useful thing is that a Mode statement (line 30) automatically clears the screen when a program is run. The awkward thing is that, because the program runs in Mode 5 it *lists* in Mode 5 as well. This is awkward because Mode 5 text is difficult to read. To get round this type the following:

```
MODE 7

LIST
```

and there is your program listed in Mode 7 (it still runs in Mode 5). But you can go one better than that by setting two of the user-defined keys (the red ones) before you start your graphics work. Type the following using the black keys:

```
*KEY 0 RUN |M (press Return)
*KEY 9 MODE7 |M LIST |M (press Return)
```

(| is shifted \; it appears as ÷ when the screen is in Mode 7. |M creates a carriage return).

Now, whenever you press red key `f0`, the program runs and whenever you press red key `f9` it lists in Mode 7. Check that this works on Program 4.2. (You may wish to add these user-defined key commands as additional lines in the programs in this unit so that the keys will be set up whenever you run the programs).

One last preliminary. When using graphics, we don't want the text cursor flashing away all the time so we turn it off with the statement `VDU 23; 8202; 0; 0; 0;`. Use `VDU 23, &670A; 0; 0; 0;` to turn it on again.

4.3 DRAW

The first graphics statement that we will introduce is

```
DRAW X, Y
```

This means draw a line from wherever the graphics cursor now is to the point (*X*, *Y*). (You can't see the graphics cursor but it is the point on the screen at which the next graphics item will appear). The graphics cursor returns to (0, 0) whenever a Mode change is given, so

```
10 REM Drawing a line
20
30 MODE 5
40 GCOL 0, 2
50 DRAW 200, 500
```

Program 4.3

gives Figure 4.2.

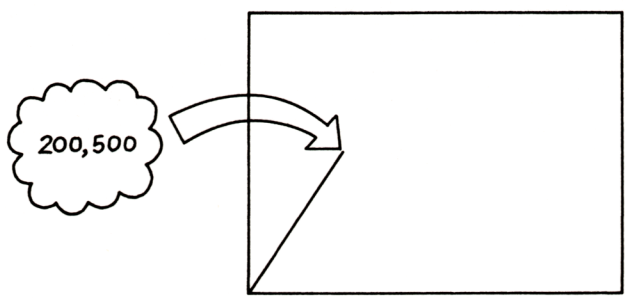


Figure 4.2

SAQ 1

Draw (with pencil and paper) the result of:

```
20 MODE 5
30 DRAW 200, 500
40 DRAW 300, 500
50 DRAW 400, 900
60 DRAW 800, 0
```

Program 4.3a

SAQ 2

Write a program to draw Figure 4.3.

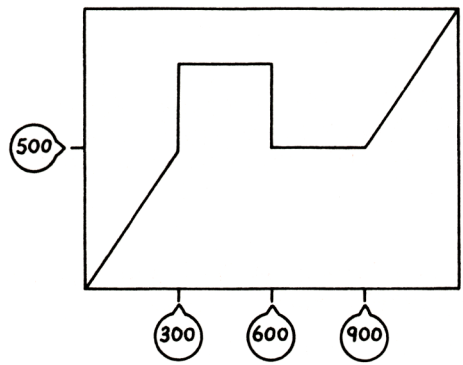


Figure 4.3

4.4 MOVE

It's a bit awkward having to draw from wherever we last were and for that reason there is another statement:

```
MOVE X, Y
```

which moves the graphics cursor to (*X*, *Y*) without anything appearing on the screen. So Program 4.4 results in Figure 4.4.

```
10 REM Moving without drawing
20
30 MODE 5
40 GCOL 0, 5
50 MOVE 300, 510
60 DRAW 900, 510
```

Program 4.4

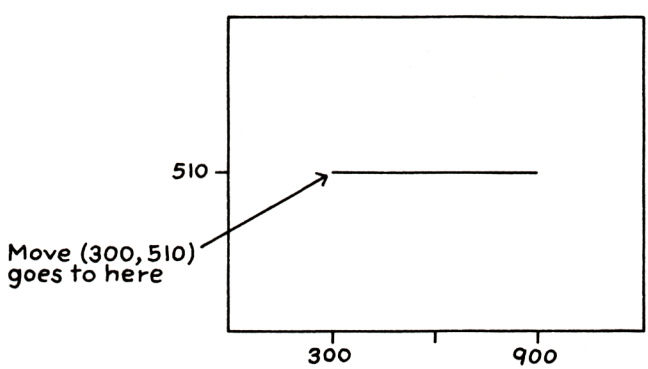


Figure 4.4

SAQ 3

Write a program to draw a box 600 points wide and 400 points high in the middle of the screen.

4.5 PLOT 85

As you will shortly see, PLOT is a versatile statement which does an enormous range of things depending on the number that follows it. But one PLOT statement is especially useful so we are giving it this separate section.

```
PLOT 85, X, Y
```

draws a triangle between the point (X, Y) and the last two points visited on the screen and fills it with colour. Program 4.5 produces Figure 4.5.

```
10 REM Filling a triangle
20
30 MODE 5
40 VDU 23;8202;0;0;0;
50 GCOL 0, 2
60
70 REM Move to first and second points
80 MOVE 800, 600
90 MOVE 800, 900
100
110 REM Fill in triangle to third point
120 PLOT 85, 200, 200
```

Program 4.5

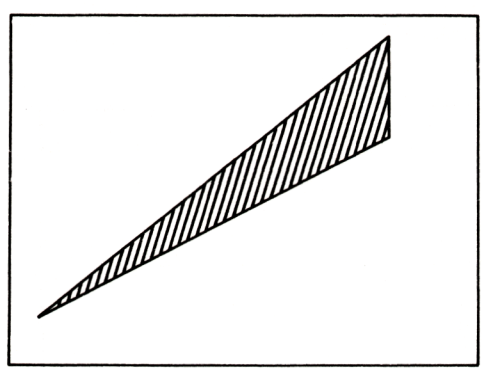


Figure 4.5

K Type and run Program 4.5.

SAQ 4

Amend two lines of the answer to SAQ 3 in order to fill the rectangle with colour.

SAQ 5

Write a program to draw Figure 4.6 and to fill the rectangle with colour.

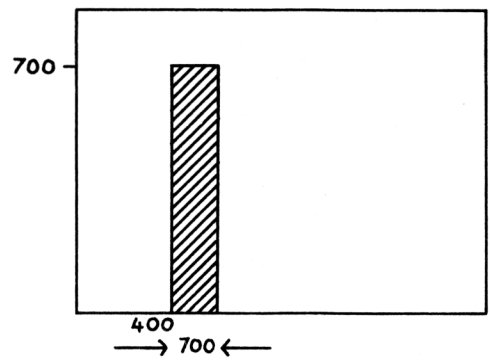


Figure 4.6

SAQ 5 may look very trivial but you may also have noticed that it looks very like a histogram (or bar chart) block. If we can draw one block, why not all the blocks for a complete histogram?

Example 1

Write a program to draw a single colour histogram for the following data:

X	Frequency F(X)
0 - 99	100
100 - 199	200
200 - 299	250
300 - 399	700
400 - 499	500
500 - 599	50
600 - 699	25
700 - 799	0
800 - 899	75
900 - 999	150

Solution

Whilst this is not difficult, it is certainly sufficiently troublesome to justify our wanting to make the program as general as possible. We will therefore write the block drawing routine as a procedure which

requires specification of the number of blocks to be drawn (N) and the block width (W).
The structure will be:

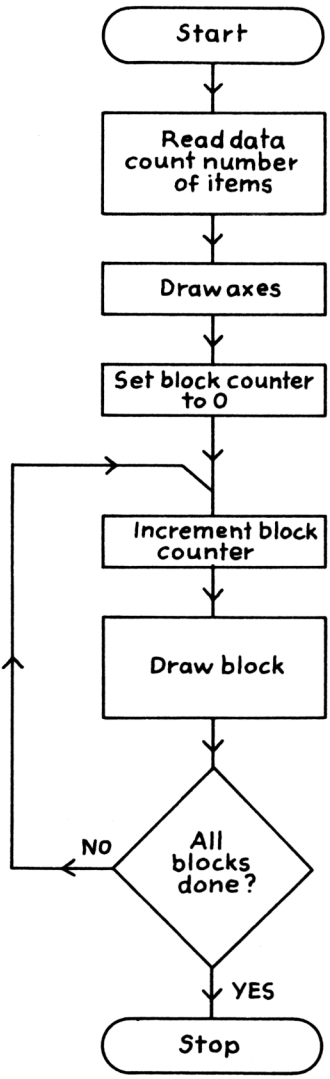


Figure 4.7 Histogram flowchart

The block drawing and colouring routine is as in SAQ 5, i.e.

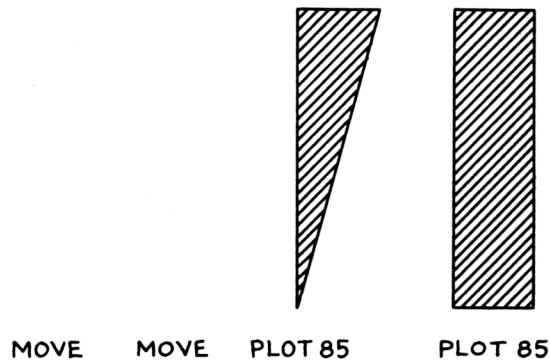


Figure 4.8 Steps to draw and fill a histogram block

For each block, we move to a new starting position and for each block the height is equal to the frequency for that block. (Frequency means the number of times we have counted something as occurring.) The program is:

```
 10 REM Histogram plotter
 20
 30 MODE 5
 40 GCOL 0, 2
 50 DIM F(30) : REM holds frequencies
 60
 70 REM Read frequencies into array
 80 N = 0
 90 REPEAT
100   N = N + 1
110   READ F(N)
120   UNTIL F(N) = - 9999
130 N = N - 1
140
150 PROChistogram(N, 100)
160 END
170
180 DEF PROChistogram(N, W)
190 REM Draw axes
200 MOVE 0, 0
210 DRAW 1200, 0
220 MOVE 0, 0
230 DRAW 0, 800
240
```

```
250 REM Draw blocks
260 FOR I = 1 TO N
270   PROCblock(I, W, F(I))
280   NEXT I
290
300 ENDPROC
310
320 DEF PROCblock(I, W, F(I))
330
340 MOVE (I - 1) * W, 0
350 MOVE (I - 1) * W, F(I)
360 PLOT 85, I * W, 0
370 PLOT 85, I * W, F(I)
380
390 ENDPROC
400
410 DATA 100, 200, 250, 700, 500, 50, 25, 0, 75, 156
420 DATA -9999
```

Program 4.6

The resulting picture is Figure 4.9.

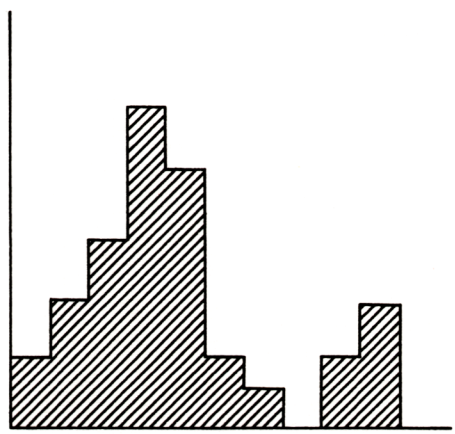


Figure 4.9 The single-colour histogram

K Type and run Program 4.6.

And now, clearly, we need to look at ways of getting more colours onto the screen.

4.6 Colour selection 1

We shall introduce the colour control system by some exploratory programs.

K Run Program 4.7 to see the effect of changing colours. We shall then go on to explain the use of `COLOUR` which is the item of interest in this program:

```
10 REM Mode 5 colours
20
30 MODE 5
40 COLOUR 0
50 PRINT "HELLO"
60 COLOUR 1
70 PRINT "HELLO"
80 COLOUR 2
90 PRINT "HELLO"
100 COLOUR 3
110 PRINT "HELLO"
```

Program 4.7

You would expect to see `HELLO` on the screen four times but you only see it three times. This is because colour 0 is black and, as the program stands, the screen background is black as well.

K Type and run Program 4.7a.

```
10 MODE 5
20 COLOUR 2
30 PRINT "HELLO"
40 DRAW 700, 300
```

Program 4.7a

`HELLO` is in colour 2 (yellow in mode 5) but the graphics are white. Now insert `22 GCOL 0, 2`. This time the graphics are yellow.

Now add `24 COLOUR 129`. The *yellow* text is now on a *red* background.

And finally, add line `26 GCOL 0, 130` with line `28 CLG`. The screen fills with yellow as a background yet the text retains its own red background. Naturally the line (which was yellow) has disappeared against the yellow screen. (Change `22 GCOL 0, 2` to `22 GCOL 0, 1` to plot the line in red and it will reappear.)

You can see from this that the BBC Micro has four separate types of area on the screen:

text	(= text foreground)
text background	
graphics	(= graphics foreground)
graphics background	

The colours of these are set by:

Text	`COLOUR` Number between 0 and 16
Text background	`COLOUR` Number between 128 and 143 (= 128 + (0 to 16))
Graphics	`GCOL 0` Number between 0 and 16
Graphics background	`GCOL 0` Number between 128 and 143 (= 128 + (0 to 16))

If you have tried getting colours at random you will have noticed that only certain numbers produce colours – others have no effect at all. That is because each mode can display only a certain number of colours:

Mode 0	2 colours)	
Mode 1	4 colours)	
Mode 2	16 colours)	Model B only
Mode 3	2 colours)	
Mode 4	2 colours	
Mode 5	4 colours	
Mode 6	2 colours	
Mode 7	7 colours	

Table 4.1 Modes and colours

The colours which you get from a colour number vary. Thus:

2 = yellow foreground in Modes 1 and 5
2 = green in Mode 2

And

130 = yellow background in Modes 1 and 5
130 = green background in Mode 2.

	Model B only				**Models A and B**			
Colour	Mode 0	Mode 1	Mode 2	Mode 3	Mode 4	Mode 5	Mode 6	Mode 7*
Black	0(128)	0(128)	0(128)	0(128)	0(128)	0(128)	0(128)	-
Red	-	1(129)	1(129)	-	-	1(129)	-	129
Green	-	-	2(130)	-	-	-	-	130
Yellow	-	2(130)	3(131)	-	-	2(130)	-	131
Blue	-	-	4(132)	-	-	-	-	132
Magenta	-	-	5(133)	-	-	-	-	133
Cyan	-	-	6(134)	-	-	-	-	134
White	1(129)	3(131)	7(135)	1(129)	1(129)	3(131)	1(129)	135

Flashing colours

Black-white	8(136)
Red-cyan	9(137)
Green-magenta	10(138)
Yellow-blue	11(139)
Blue-yellow	12(140)
Magenta-green	13(141)
Cyan-red	14(142)
White-black	15(143)

( ) = background numbers

Table 4.2 BBC Micro colour numbers

*Mode 7 colours work differently from all the other modes. You do *not* use statements like `COLOUR` or `GCOL 0`. What you have to use is:

```
PRINT CHR$(colour number) "item to be printed".
```

This is demonstrated by the following program segment:

```
10 MODE 7
20 FOR I = 0 TO 7
30   PRINT CHR$(128 + I) "MODE 7 COLOURS"
40   NEXT I
```

(`CHR$` stands for '*character string*'. This is explained in more detail in Unit 6).

Example 2

Write a program in Mode 5 to print `"THIS IS THE MODE 5 COLOUR RANGE"` in red text on a yellow background. Place the text on a white screen and surround it with a black rectangle.

Solution

```
 10 REM Mode 5 colour range
 20
 30 MODE 5
 40 VDU 23;8202;0;0;0;
 50 COLOUR 1 : REM Red text
 60 COLOUR 130 : REM Yellow text background
 70 GCOL 0, 0 : REM Black graphics
 80 GCOL 0, 131 : REM White graphics background
 90 CLG : REM Covers screen with graphics background
    colour
100
110 PRINT TAB(3, 10) "THIS IS THE "
120 PRINT TAB(3, 11) "   MODE 5   "
130 PRINT TAB(3, 12) "COLOUR RANGE"
```

```
140
150 MOVE 100, 400
160 DRAW 100, 800
170 DRAW 1100, 800
180 DRAW 1100, 400
190 DRAW 100, 400
```

Program 4.8

Notice the extra spaces inside the quotes of lines 110 and 120. These are needed to print yellow and so display the message neatly.

K Type and run Program 4.8.

SAQ 6

Write a program in Mode 5 to print a message in black letters on a yellow background. Surround the message with a white triangle on a red background.

4.7 Colour selection 2

So far, we have always specifically written the number(s) of the colour(s) we want into each program. But sometimes we want the program flow itself to choose the colour. That means the colour number has to be a variable. If you look back to Program 4.7, we had to write 4 sets of instructions to print 4 lines in 4 different colours. Why not use a `FOR ... NEXT` loop?

```
10 REM Mode 5 colour range
20
30 MODE 5
50 FOR I = 0 TO 3
60   COLOUR I
70   PRINT "HELLO"
80   NEXT I
```

Program 4.9

SAQ 7

Write a program to print ten boxes, each containing the ones previous to it. Make the first box red, second yellow, third white, fourth red, etc.

We are now ready to return to the histogram. We left it with all the blocks in yellow and the whole picture set against a black background. How much better if the blocks went red, yellow, white, red, yellow, white ... so that each stands out clearly from its neighbour.

SAQ 8

Modify Program 4.6 to rotate the colours red, yellow and white through the blocks.

SAQ 9

Write a program to plot triangles of random shape and size at random positions on the screen with their colour chosen at random from black, red, yellow and white. (Here, plotting in black *does* make sense, as you will see).

Even more colours

If you have had enough of yellow, red and white (especially if you are using a Model A), you can use other colours with the `VDU 19` command. This command allows you to change the colours which a Mode normally gives you. (You can't use it in Mode 7). You have seen that, say, Mode 4 is a two-colour Mode and Mode 5 is a four-colour Mode. Well, nothing can stop that, but with `VDU 19` you can change which two (or four) colours you get. You use it as follows:

```
VDU 19,2,6,0,0,0
old colour ↗   ↖ new colour
```

changes colour 2 in your present Mode to colour 6 in Mode 2.

(An alternative form for `VDU 19, 2, 6, 0, 0 , 0` is `VDU 19, 2, 6; 0;`).

SAQ 10

What effect do these `VDU 19` statements have?

(a) `VDU 19, 0, 6, 0, 0, 0` in Mode 4
(b) `VDU 19, 0, 11, 0, 0, 0` in Mode 5
(c) `VDU 19, 2, 2, 0, 0, 0` in Mode 5
(d) `VDU 19, 1, 1, 0, 0, 0` in Mode 4
(e) `VDU 19, 3, 4, 0, 0, 0` in Mode 5

SAQ 11

Write statements to:

(a) change the normal range of Mode 5 colours to red, green, yellow and blue;
(b) change the normal range of Mode 4 colours to magenta and green.

4.8 Text and graphics

Occasionally (e.g. SAQ 6) we have mixed text and graphics by using `PRINT TAB(X,Y)` as usual. However, this gives a relatively crude placing of the print on the graphics and there are many occasions when we wish to position text more precisely. This is most likely to be the case in Modes 0, 1 or 4 where the graphics resolution is very high. On the BBC Micro you can position text as accurately as graphics by the use of the statement `VDU 5`. This 'locks' together the text and graphics cursors

so that the text cursor follows the graphics co-ordinates. The text is then printed in the graphics foreground colour. Program 4.10 demonstrates this.

```
10 REM Printing with TAB and MOVE
20
30 MODE 5
40 GCOL 0, 1
50 PRINT TAB(10, 5) "HERE"
60 VDU 5
70 MOVE 200, 400 : PRINT "HERE"
```

Program 4.10

[K] Type and run Program 4.10.

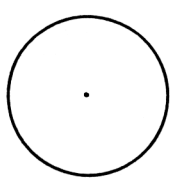
After using `VDU 5` to join the text and graphics cursors together, you may wish to use them separately. You can separate them again with the statement `VDU 4`. One odd point about printing text with VDU 5 is that you can't erase one word just by printing another of the same number of letters on top – all you get is a mixture of the two words.

We are now in a position to produce a useful application of graphics and text.

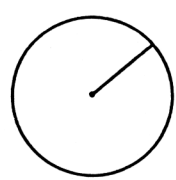
Example 3
Write a program to produce a pie chart display with three sections: 40%, 30% and 30%.

Solution
The solution to this requires either (a) the use of some trigonometry and co-ordinate geometry or (b) Pythagoras' theorem. We have chosen the first of these two methods.

The procedure we use is to work in 0.5 percent sectors, each of which is represented by a turn of 1.8 degrees. So, to produce the 40 percent sector we need to make 80 moves, turning one sector anti-clockwise each time:



MOVE



PLOT 85

The result is a triangular sector (except after the first `MOVE / PLOT 85` pair) but it gives a reasonably circular result on the screen. This is best drawn in polar co-ordinates. First, we make life easier for ourselves by moving the graphics origin to the centre of the screen.

Moving the graphics origin

The statement:

```
VDU 29,600;500;
```

comma

semi-colons

moves the graphics origin to the point (600, 500). The syntax of this is important so note the position of the comma and two semi-colons. Now (0, 0) is approximately at the centre of the screen – very convenient for a pie chart – and measurements are made as follows:

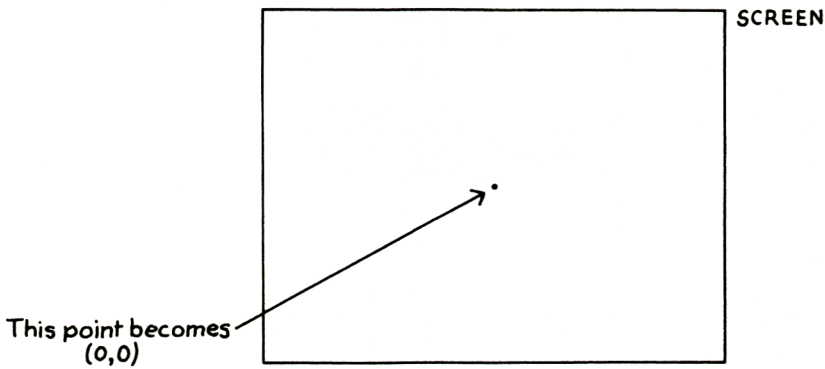


Figure 4.10 Graphics origin at the centre of the screen

If the radius of the chart is 200, the co-ordinates of any point are (`200 * COS A, 200 * SIN A`) where `A` is in radians as shown in Figure 4.11. (For a more detailed explanation, see Appendix 2.)

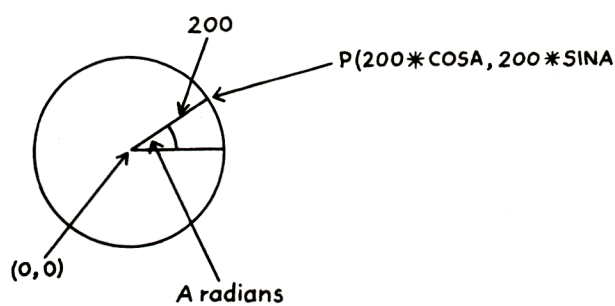


Figure 4.11 Polar co-ordinates

The flowchart for the algorithm is shown in Figure 4.12. The program is Program 4.12.

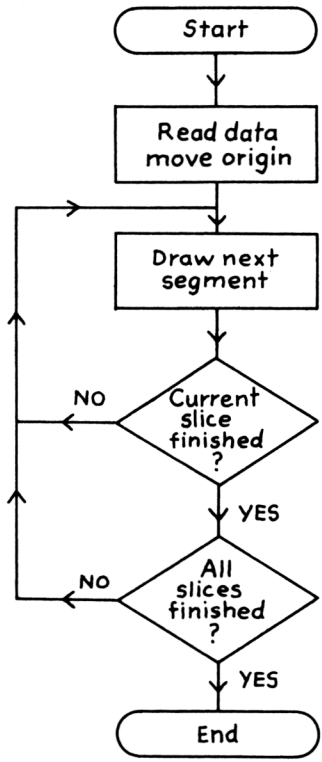


Figure 4.12 Flowchart for pie chart drawing

```
 10 REM Pie chart
 20
 30 MODE 5
 40 VDU 23;8202;0;0;0;
 50 READ slices
 60 DIM piece(slices)
 70
 80 FOR I = 1 TO slices
 90   READ piece(I)
100   NEXT I
110
120 REM Move origin
130 VDU 29, 600 ; 500 ;
140 stepsize = RAD(1.8)
150 radius = 300
```

```
160 start = 0
170
180 REM Draw slices
190 FOR I = 1 TO slices
200   GCOL 0, (I MOD 3) + 1
205   REM piece(I) * 3.6 converts slice percentage to an angle
210   finish = start + RAD(piece(I) * 3.6) - stepsize
220
230   REM Draw a thin triangle
240   MOVE COS(start) * radius, SIN(start) * radius
245
250   FOR angle = start TO finish STEP stepsize
260     MOVE 0,0
270     PLOT 85, COS(angle  + stepsize)  * radius,  SIN(angle +
        stepsize) * radius
280     NEXT angle
290
300   start = finish + stepsize
310   NEXT I
320
330 END
340
350 DATA 3, 40, 30, 30
```

Program 4.11

K Type and run Program 4.11.

SAQ 12

Modify Program 4.12 to make the display as shown in Figure 4.13.

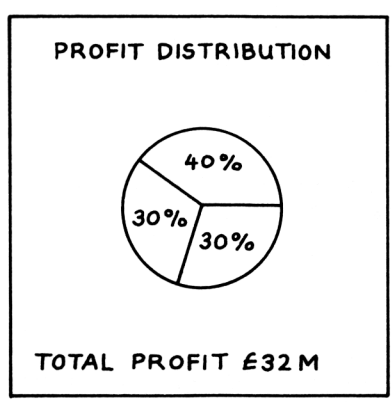


Figure 4.13 Pie chart with text

4.9 Text and graphics windows

There are occasions when you want to be able to use the screen to display graphics and normal text i.e. text not displayed on the graphics. This can be done by two VDU statements which allow you to choose which area of the screen is to display text and which graphics.

K To see this in operation, enter Program 4.12.

```
 10 REM Regular figures
 20
 30 MODE 4
 40 VDU 19, 1, 3, 0, 0, 0 : REM White as yellow
 50 REM Define text area
 60 VDU 28, 5, 30, 35, 28
 70 REM Define graphics area
 80 VDU 24, 0 ; 200 ; 1200 ; 1000 ;
 90 radius = 400
100
110 REPEAT
120   CLS
130   PRINT "Regular figures."
140   INPUT "Enter No. of sides " sides
150
160   CLG
170   REM Choose random location for centre of figure
180   X = RND(400) + 200
190   Y = RND(500) + 300
200   sector = 2 * PI / sides
210
220   REM Draw figure
230   MOVE X + radius * COS(sector), Y + radius * SIN(sector)
240   FOR I% = 0 TO sides
250     DRAW X + radius * COS(I% * sector), Y + radius
        * SIN(I% * sector)
260     NEXT I%
270
280   CLS
290   INPUT "Another figure? (Y/N) " A$
300   UNTIL A$ <> "Y"
310
320 END
```

Program 4.12

The text is always printed in a 'text window' 3 lines high at the bottom of the screen and the graphics in a graphics window which starts 200 points from the bottom of the screen. When the figure is too big for the area allotted to the graphics, it is cut off by the edge of the window. We

have exaggerated this effect in Program 4.12 by drawing the figures with a radius of 400. The program looks better if you use a smaller radius.

Text and graphics windows are illustrated in Figure 4.14.

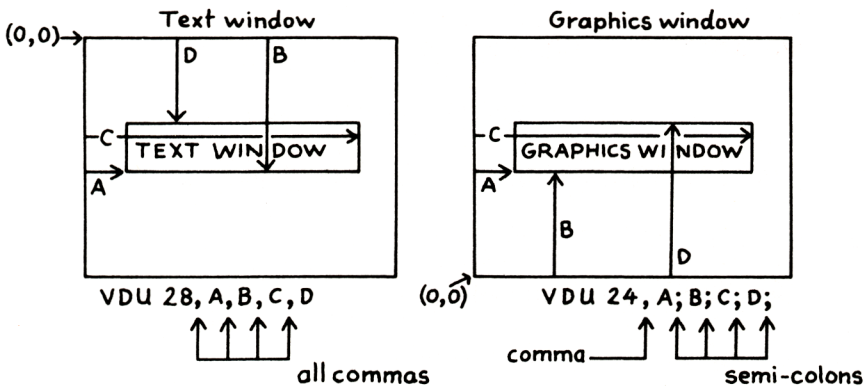


Figure 4.14 Defining text and graphics windows

If at a later stage you want to remove the windows, you can do so with `VDU 26` which cancels *both* windows *and* restores the graphics origin to the left-hand bottom corner.

SAQ 13

Sketch the windows defined by the following:

(a) In Mode 4: `VDU 28, 10, 4, 29, 0`
(b) In Mode 4: `VDU 28, 0, 16, 39, 16`
(c) In Mode 5: `VDU 28, 15, 20, 19, 12`
(d) `VDU 24, 0; 500; 1279; 1023;`
(e) `VDU 24, 0; 0; 600; 1023;`
(f) In Mode 4: `VDU 24, 0; 0; 700; 1023; : VDU 28, 25, 20, 35, 10`

4.10 PLOT n, X, Y

You have already met `PLOT 85, X, Y` which draws triangles and fills them with the current graphics foreground colour. This is one particular case of

```
PLOT n, X, Y
```

The BBC Micro produces an enormous range of effects from `PLOT n, X, Y`, depending on the value of n. We shall now briefly introduce 48 such effects. There are 6 basic effects and 8 variations on each. The 6 are shown in Table 4.3.

n	PLOT n, X, Y produces
0 - 7	a line
8 - 15	a line less the last point
16 - 26	a dotted line
24 - 31	a dotted line less the last point
64 - 71	a point at X, Y
80 - 87	fills a triangle

Table 4.3 Effect of n in PLOT n, X, Y

The choice of number *within* the range above determines

(a) the colour that PLOT works in
(b) whether the action is relative to the current graphics origin or relative to the last point visited.

Table 4.4 summarises all the possibilities:

	Values of n in PLOT n, X, Y					
Type of result	Line	Line less last point	Dotted line	Dotted line less last point	Point	Fill a triangle
Relative to last point						
Move	0	8	16	24	64	80
Draw in gf colour	1	9	17	25	65	81
Draw in li colour	2	10	18	26	66	82
Draw in gb colour	3	11	19	27	67	83
Relative to origin						
Move	4	12	20	28	68	84
Draw in gf colour	5	13	21	29	69	85
Draw in li colour	6	14	22	30	70	86
Draw in gb colour	7	15	23	31	71	87

Key

gf = graphics foreground
li = logical inverse (li of colour number C is number of colours in MODE – 1 – C) (not used in this course)
gb = graphics background

Table 4.4 The full range of effects of PLOT n, X, Y

Notice that

```
PLOT 4, X, Y is the same as MOVE X, Y
```

and

```
PLOT 5, X, Y is the same as DRAW X, Y
```

The following program demonstrates some effects from `PLOT n, X, Y`.

SAQ 14

Sketch what you would expect to see as a result of running this program. Then check on the screen and against the answer at the end of the unit.

```
 10 REM Plot test
 20
 30 MODE 5
 40 GCOL 0, 1
 50 GCOL 0, 130
 60 VDU 19, 3, 1, 0, 0, 0
 70 CLG
 80
 90 PLOT 4, 300, 300
100 PLOT 5, 300, 800
110 PLOT 5, 800, 800
120 PLOT 5, 800, 300
130 PLOT 5, 300, 300
140
150 MOVE 400, 400
160
170 PLOT 21, 400, 700
180 PLOT 21, 700, 700
190 PLOT 21, 700, 400
200 PLOT 21, 400, 400
210
220 FOR I = 100 TO 1100 STEP 100
230   PLOT 69, I, 100
240   NEXT I
```

Program 4.13

Example 4

Write a program to draw a clock on the screen which shows second, hour and minute hands and keeps accurate time.

We have covered a great deal in this unit, and this is a fairly ambitious example to pull together quite a few of the ideas. First the flowchart is shown in Figure 4.15.

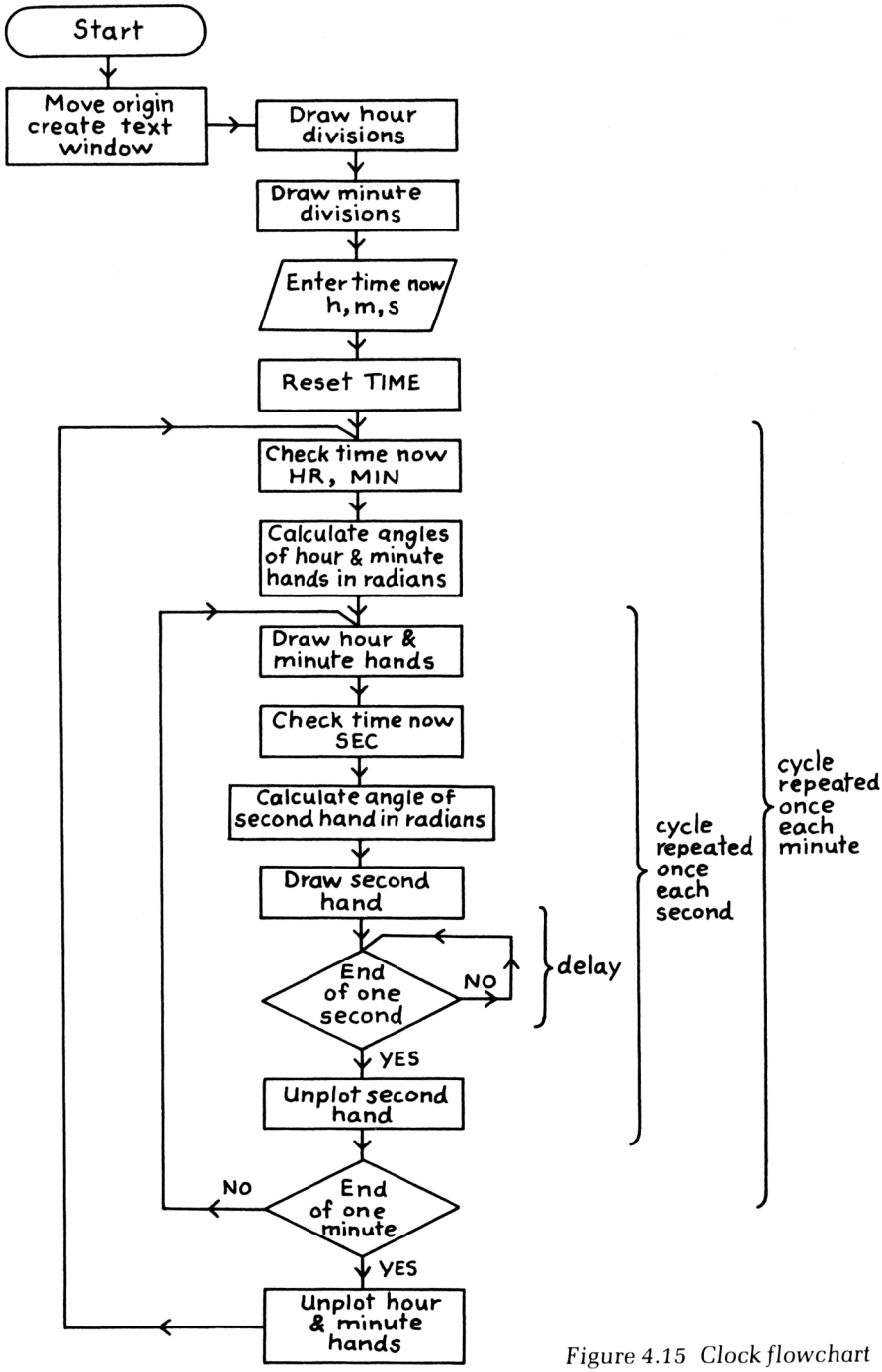


Figure 4.15 Clock flowchart

The program is discussed below.

- First, move the origin to the centre of the screen and create a text window for messages to the user.

```
10 REM Clock
20
30 REM Move origin to centre of screen/create text window
40 MODE 5
50 VDU 29, 600 ; 500 ;
60 VDU 28, 0, 31, 19, 28
```

Program 4.14a

- Second, draw a clock face. The first loop (lines 90 – 120) draws the hour divisions using `PLOT 68` which *moves* the cursor out 300 points to the edge of the clock. Then it uses `PLOT 1` to draw a line 30 points long at each hour position on the clock. The second loop (lines 140 – 170) inserts minute divisions in a similar way but using `PLOT 17` to give dotted divisions.

```
 80 REM Draw clock face
 90 FOR angle = 0 TO 2 * PI  STEP PI / 6
100   PLOT 68, 300 * COS(angle), 300 * SIN(angle)
110   PLOT 1, 30 * COS(angle), 30 * SIN(angle)
120   NEXT angle
130
140 FOR angle = 0 TO 2 * PI STEP PI / 30
150   PLOT 68, 270 * COS(angle), 270 * SIN(angle)
160   PLOT 17, 20 * COS(angle), 20 * SIN(angle)
170   NEXT angle
```

Program 4.14b

- Third, we have to synchronise the computer's time with the user's time. We convert the correct time in the outside world to hundredths of a second and then re-set the computer's clock (line 230) to this value.

```
190 REM Set time
200 INPUT "Hour now    " hour
210 INPUT "Minutes now " minutes
220 INPUT "Seconds now " seconds
230 TIME = 100 * seconds + 6000 * minutes + 360000 * hour
```

Program 4.14c

☐ Fourth, the clock is run on the screen. There are many ways of doing this. We have chosen to

(a) move the second hand once each second;
(b) move the minute and hour hands once each minute.
The loop (lines 360 – 550) moves the second hand.

```
340   REPEAT
350
360     REM Hours and minutes
370     REM Drawn once each second
380     MOVE 0, 0
390     GCOL 0, 2
400     DRAW 200 * COS(H), 200 * SIN(H)
410     MOVE 0, 0
420     DRAW 250 * COS(M), 250 * SIN(M)
430
440     REM Seconds
450     SEC = (TIME DIV 100) MOD 60
460     S = PI / 2 - SEC * PI / 30
470     MOVE 0, 0
480     GCOL 0, 1
490     DRAW 250 * COS(S), 250 * SIN(S)
500     REPEAT
505       UNTIL TIME MOD 100 = 0
510     MOVE 0,0
520     GCOL 0, 0
530     DRAW 250 * COS(S), 250 * SIN(S)
540
550     UNTIL (TIME DIV 100) MOD 60 = 0
```

Program 4.14d

The purpose of this loop is to move and re-draw the second hand once every second. But because the second hand moves 'over' the minute and hour hands we have to re-draw them (lines 390 - 420) as well. If we don't do this, they disappear when the second hand passes 'over' them.

☐ Fifth, the rest of the program is the outer `REPEAT ... UNTIL` loop which encloses the second hand drawing loop. It consists of:

```
270 REPEAT
280   REM Draws once every minute
290   HR = (TIME DIV 360000) MOD 12
300   MIN = (TIME DIV 6000) MOD 60
310   H = PI / 2 - HR * PI / 6 - (PI / 360) * (MIN MOD 60)
320   M = PI / 2 - MIN * PI / 30
330
```

```
...
...
...
  560
  570   GCOL 0,0
  580   MOVE 0, 0 : DRAW 250 * COS(M), 250 * SIN(M)
  590   MOVE 0, 0 : DRAW 200 * COS(H), 250 * SIN(H)
  600
  610   UNTIL FALSE
```

Program 4.14e

The full program is:

```
  10 REM Clock
  20
  30 REM Move origin to centre of screen/create text window
  40 MODE 5
  50 VDU 29, 600 ; 500 ;
  60 VDU 28, 0, 31, 19, 28
  70
  80 REM Draw clock face
  90 FOR angle = 0 TO 2 * PI  STEP PI / 6
 100   PLOT 68, 300 * COS(angle), 300 * SIN(angle)
 110   PLOT 1, 30 * COS(angle), 30 * SIN(angle)
 120   NEXT angle
 130
 140 FOR angle = 0 TO 2 * PI STEP PI / 30
 150   PLOT 68, 270 * COS(angle), 270 * SIN(angle)
 160   PLOT 17, 20 * COS(angle), 20 * SIN(angle)
 170   NEXT angle
 180
 190 REM Set time
 200 INPUT "Hour now    " hour
 210 INPUT "Minutes now " minutes
 220 INPUT "Seconds now " seconds
 230 TIME = 100 * seconds + 6000 * minutes + 360000 * hour
 240
 250 REM Run clock on screen
 260 CLS
 270 REPEAT
 280   REM Draws once every minute
 290   HR = (TIME DIV 360000) MOD 12
 300   MIN = (TIME DIV 6000) MOD 60
 310   H = PI / 2 - HR * PI / 6 - (PI / 360) * (MIN MOD 60)
 320   M = PI / 2 - MIN * PI / 30
 330
```

```
340   REPEAT
350
360     REM Hours and minutes
370     REM Drawn once each second
380     MOVE 0, 0
390     GCOL 0, 2
400     DRAW 200 * COS(H), 200 * SIN(H)
410     MOVE 0, 0
420     DRAW 250 * COS(M), 250 * SIN(M)
430
440     REM Seconds
450     SEC = (TIME DIV 100) MOD 60
460     S = PI / 2 - SEC * PI / 30
470     MOVE 0, 0
480     GCOL 0, 1
490     DRAW 250 * COS(S), 250 * SIN(S)
500     REPEAT
505       UNTIL TIME MOD 100 = 0
510     MOVE 0,0
520     GCOL 0, 0
530     DRAW 250 * COS(S), 250 * SIN(S)
540
550     UNTIL (TIME DIV 100) MOD 60 = 0
560
570   GCOL 0,0
580   MOVE 0, 0 : DRAW 250 * COS(M), 250 * SIN(M)
590   MOVE 0, 0 : DRAW 200 * COS(H), 250 * SIN(H)
600
610   UNTIL FALSE
```

Program 4.14

[K] Type and run Program 4.14.

SAQ 15

We left the histogram example in SAQ 8 after rotating the colours between the blocks. Now try adding three further refinements to it.

First, improve the colour range by making the yellow print as green and the white as blue.

Second, put in a system for automatic y-axis scaling.

Third, let the block width be chosen according to the number of blocks instead of being fixed at 100. Make it so that the highest block is always 0.75 of the height of screen, whatever the frequency of the highest block. (You will have to insert in the program a routine to detect the highest frequency).

Assignment D

1. Make a program to produce a weather map of the United Kingdom something as in Figure 4.16. (Make the outline as crude or refined as you care to). Print weather symbols for 'rain', 'sun' and 'snow' at various points on the map.

Insert some relevant message in the text window, such as temperatures and wind speeds.

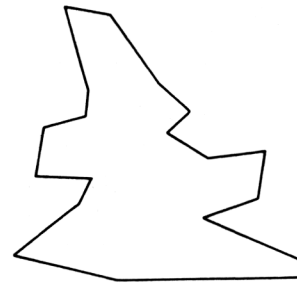


Figure 4.16

Objectives of Unit 4

By the end of this unit you should be able to use:

- ☐ `MOVE X, Y` to move the graphics cursor.
- ☐ `PLOT 85, X, Y` to fill triangles with colour.
- ☐ `COLOUR ...` to select text foreground and background colours.
- ☐ `GCOL 0, ...` to select graphics foreground and background colours.
- ☐ `VDU 19, ...` to change a Mode's normal colour range.
- ☐ `VDU 5` to join text and graphics cursors.
- ☐ `VDU 4` to separate text and graphics cursors.
- ☐ `VDU 28, ...` to define a text window.
- ☐ `VDU 24, ...` to define a graphics window.
- ☐ `VDU 29, ...` to move the graphics origin.
- ☐ `VDU 26` to restore default text and graphics.
- ☐ `PLOT n, X, Y` to produce a variety of results, e.g. dotted lines.

Answers to SAQs

SAQ 1

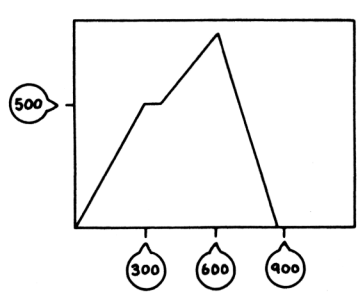


Figure 4.17

SAQ 2

```
10 REM Draw demonstration
20
30 MODE 5
40 GCOL 0, 2
50 DRAW 300, 400
60 DRAW 300, 700
70 DRAW 600, 700
80 DRAW 600, 400
90 DRAW 900, 400
100 DRAW 1279, 1023
```

Program 4.15

Don't worry if you had slightly different numbers as long as your program produced the general effect and made the vertical lines vertical and the horizontal lines horizontal.

K Type and run Program 4.15.

SAQ 3

```
10 REM Draw a box
20
30 MODE 5
40 GCOL 0, 2
50 MOVE 300, 300
60 DRAW 300, 700
70 DRAW 900, 700
80 DRAW 900, 300
90 DRAW 300, 300
```

Program 4.16

K Type and run Program 4.16.

SAQ 4

Change lines 80 and 90 to:

```
80 PLOT 85, 300, 300
90 PLOT 85, 900, 300
```

SAQ 5

```
10 REM Draw a block
20
30 MODE 5
40 GCOL 0, 2
50 MOVE 400, 0
60 MOVE 400, 700
70 PLOT 85, 500, 0
80 PLOT 85, 500, 700
```

Program 4.17

K Type and run Program 4.17.

SAQ 6

```
10 REM Triangle and message
20
30 MODE 5
40 COLOUR 0
50 COLOUR 130
60 GCOL 0, 3
70 GCOL 0, 129
80 CLG
90
100 REM Print the text
110 PRINT TAB(7, 15)  "THIS"
120 PRINT TAB(8, 16)  "IS"
130 PRINT TAB(8,17) " A "
140 PRINT TAB(5, 18)  "TRIANGLE"
150
160 REM Draw the triangle
170 MOVE 100, 100
180 DRAW 1100, 100
190 DRAW 550, 900
200 DRAW 100, 100
```

Program 4.18

K Type and run Program 4.18.

SAQ 7

```
10 REM A set of boxes
20
30 MODE 5
40 VDU 23;8202;0;0;0;
50 colour = 0
60 boxes = 10
70
80 FOR I = 1 TO boxes
90
100   REM Colour rotation
110   REM Don't use 0 (= black)
120   colour = colour + 1
130   colour = colour MOD 3 + 1
140   GCOL 0, colour
150
160   REM Draw boxes
170   J = (I -1) * 50
180   MOVE 400 - J, 400 - J
```

```
190   DRAW 400 - J, 600 + J
200   DRAW 800 + J, 600 + J
210   DRAW 800 + J, 400 - J
220   DRAW 400 - J, 400 - J
230
240   NEXT I
```

Program 4.19

K Type and run Program 4.19.

SAQ 8

Add:

```
335 GCOL 0, (I MOD 3 + 1)
```

SAQ 9

You may have chosen to let your program run for ever, or to plot a fixed number of triangles. We have set ours to run for 100 seconds (lines 60 and 250).

```
 10 REM Random triangles
 20
 30 DIM X(3), Y(3)
 40 MODE 5
 50 TIME = 0
 60 duration = 1000
 70
 80 REPEAT
 90   REM Choose coordinates
100   FOR I = 1 TO 3
110     X(I) = RND(1200)
120     Y(I) = RND(1000)
130     NEXT I
140
150   REM Choose colour
160   colour = RND(4) - 1
170   GCOL 0, colour
180
190   REM Draw triangle
200   MOVE X(1), Y(1)
210   MOVE X(2), Y(2)
220   PLOT 85, X(3), Y(3)
230
240   UNTIL TIME > duration
```

Program 4.20

K Type and run Program 4.20.

SAQ 10

(a) Black changes to cyan.
(b) Black changes to flashing yellow-blue.
(c) Yellow changes to green. (This statement apparently changes colour 2 to colour 2. 2 in Mode 5 is yellow but in Mode 2 is green and Mode 2 numbers are the 'normal' BBC colours).
(d) White changes to red.
(e) White changes to blue.

SAQ 11

(a) Don't bother with red and yellow – they are already available in Mode 5. But you need to change black and white to green and blue. This can be done with:

```
VDU 19, 0, 2, 0, 0, 0 (black to green)
VDU 19, 3, 4, 0, 0, 0 (white to blue)
```

or with:

```
VDU 19, 0, 4, 0, 0, 0 (black to blue)
VDU 19, 3, 2, 0, 0, 0 (white to green)
```

(b)
```
VDU 19, 0, 5, 0, 0, 0 (black as magenta)
VDU 19, 1, 2, 0, 0, 0 (white as green)
```

SAQ 12

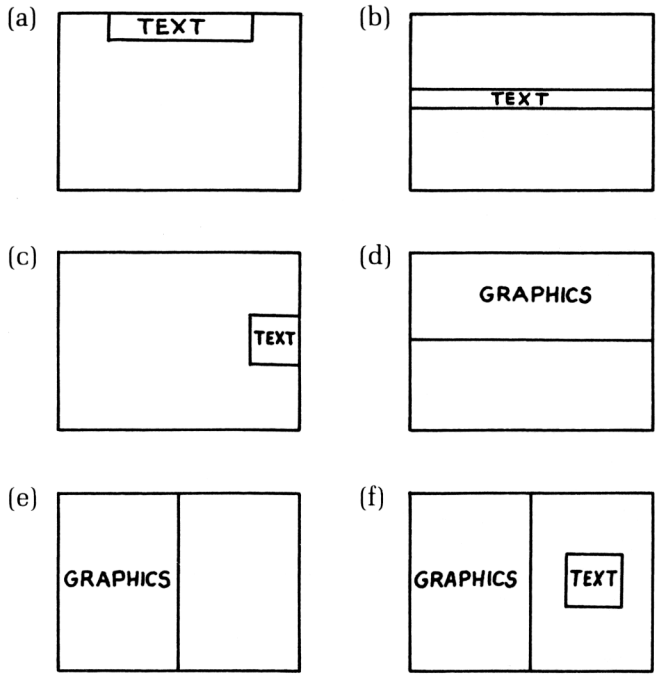
```
 10 REM Pie chart
 20
 30 MODE 5
 40 VDU 23;8202;0;0;0;
 50 READ slices
 60 DIM piece(slices)
 70
 80 PROCheading
 90 FOR I = 1 TO slices
100   READ piece(I)
110   NEXT I
120
130 REM Move origin
140 VDU 29, 600 ; 500 ;
150 stepsize = RAD(1.8)
160 radius = 300
```

```
170 start = 0
180
190 FOR I = 1 TO slices
200   GCOL 0, (I MOD 3) + 1
205   REM piece(I) * 3.6 converts slice percentage to
      an angle
210   finish = start + RAD( piece(I) * 3.6) - stepsize
220
230   MOVE COS(start) * radius, SIN(start) * radius
240   FOR angle = start TO finish STEP stepsize
250     MOVE 0,0
260     PLOT 85, COS(angle  + stepsize)  * radius,  SIN(angle +
        stepsize) * radius
270     NEXT angle
280
290   PROClabel(I)
300
310   start = finish + stepsize
320   NEXT I
330
340 END
350
360 DEF PROCheading
370
380 GCOL 0, 2
390 PRINT TAB(1, 3) "PROFIT DISTRIBUTION"
400 PRINT TAB(2, 28) "TOTAL PROFIT £32M"
410
420 ENDPROC
430
440 DEF PROClabel(I)
450
460 VDU 5
465 REM Print label at radius * .65 from centre
470 MOVE COS((start + finish) DIV 2) *  radius *  .65,
    SIN((start  + finish) DIV 2) * radius * .65
480 GCOL 0, 0
490 PRINT ; piece(I)
500
510 VDU 4
520 ENDPROC
530
540 DATA 3, 40, 30, 30
```

Program 4.21

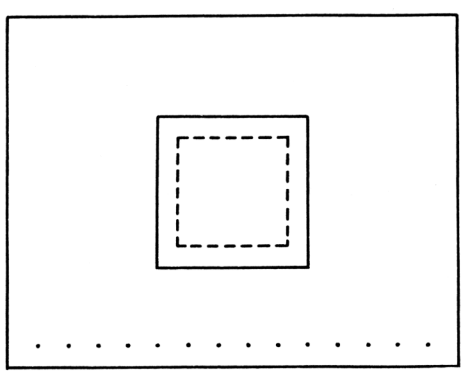
K Type and run Program 4.21.

SAQ 13



Figures 4.18(a) - 4.18(f)

SAQ 14



yellow background
red graphics

Figure 4.19

SAQ 15

```
 10 REM Histogram plotter
 20
 30 MODE 5
 40 GCOL 0, 2
 50 DIM F(30) : REM holds frequencies
 60
 70 REM Read frequencies into array
 80 N = 0
 90 REPEAT
100   N = N + 1
110   READ F(N)
120   UNTIL F(N) = - 9999
130 N = N - 1
140
150 PROChistogram(N)
160 END
170
180 DEF PROChistogram(N)
190 VDU 19, 2, 2, 0, 0, 0 : REM Yellow as green
200 VDU 19, 3, 4, 0, 0, 0 : REM White as blue
210 VDU 23;8202;0;0;0
220
230 REM Draw axes
240 MOVE 0, 0
250 DRAW 1200, 0
260 MOVE 0, 0
270 DRAW 0, 800
280
290 REM Find max frequency
300 max = 0
310 FOR I = 1 TO N
320   IF F(I) > max THEN max = F(I)
330   NEXT I
340 IF max = 0 THEN CLS : PRINT "Error. All frequencies are
    zero." : ENDPROC
350
360 REM Y axis scale factor
370 SF = .75 * (1000 / max)
380
390 REM Block width
400 W = 1000 / N
410
420 REM Draw blocks
430 FOR I = 1 TO N
```

```
440   PROCblock(I, W, F(I))
450   NEXT I
460
470 ENDPROC
480
490 DEF PROCblock(I, W, F(I))
500
510 GCOL 0, (I MOD 3 +1)
520 MOVE (I - 1) * W, 0
530 MOVE (I - 1) * W, F(I) * SF
540 PLOT 85, I * W, 0
550 PLOT 85, I * W , F(I) * SF
560
570 ENDPROC
580
590 DATA 100, 200, 250, 700, 500, 50, 25, 0
600 DATA -9999
```

Program 4.22

K Type and run Program 4.22.

UNIT 5

Program planning

5.1 Designing a program

5.2 Top-down design

5.3 Modular design

5.4 Menu driving and command driving

Assignment E

Objectives of Unit 5

Answers to SAQs

5.1 Designing a program

Because it is easy to write short BASIC programs which work, many people get into the habit of writing programs at the keyboard. What results is invariably a mess, even if it works. Good program design, on the other hand, mostly takes place on paper and should always be created away from the keyboard.

The whole process of designing and creating a program can be broken down into four stages:

Step 1 Analyse the problem
Step 2 Design a solution
Step 3 Code the solution
Step 4 Test it

Let's look at each of these steps in much more detail.

Step 1 Analyse the problem

A program is usually designed to solve a specific problem but the problem itself may be vaguely expressed. 'Write me a program to improve my stock control' might be a problem as stated to you. But what does 'improve' mean? How is the stock managed at present? What changes would be tolerated within the organisation? Clearly the problem as posed cannot be solved since it lacks precision. Any attempt to solve it must start with a careful analysis of what the user really wants. The result of this should be a clear specification for the program designer. For example, a proper analysis of the stock controller's request might reveal that his problem is (a) a lack of up-to-date stock figures and (b) a lack of a system for predicting future stock requirements.

In fact, he may well need a program which:

- ☐ Receives stock input data from keyboard entry of 'goods in'.
- ☐ Receives details of stock sold from invoice data from the computer's invoicing program.
- ☐ Prints out weekly stock figures.
- ☐ Prints out a weekly prediction of stock requirements over the next 3 months using the last 3 months (seasonally adjusted) as a basis for the prediction.

Even that analysis is only a start but it demonstrates the type of information required to design a program.

Step 2 Designing a solution

A problem will usually be stated in non-computer terms. It will be stated in the terms of its world: stock, temperatures, traffic flows. The

programmer has to see how he can design a computer-type solution to the problem. Note the phrase 'computer-type'. At this stage the programmer should not be thinking of how to write the program in a specific language, but of how to solve the problem as a series of steps. Consider, for example, the simple problem of breaking up a string of words into its separate words (see Program 3.9). A solution could be written down in ordinary English:

```
Begin
    Start next string
        Start next word
            If next character is a letter then add it to next
            word
            If next character is a space then end.
    If no more characters then end
End
```

Notice that we use indentation to denote blocks of activity which are to be repeated. At this stage we don't have to decide how to code the repetition or even the precise tests to use for ending each loop: all we are trying to do is to set out the method of solution in sufficient detail to define how it will be done. Once that has been done, the actual coding is very straightforward.

You could go on to express the solution as a flowchart as in Figure 5.1. Flowcharts can be enormously helpful for designing short sections of programs, but are not really much use for larger programs. But, as you will see, with the aid of 'top down design' and procedures, diagrams are not really needed at all.

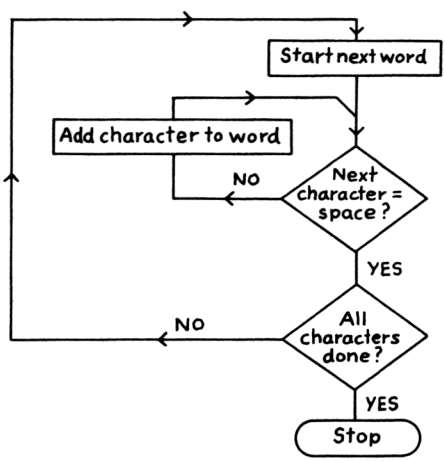


Figure 5.1 Flowchart for word splitting

Step 3 Code the solution
We don't really need to say more about this except to point out that many beginners confuse the process of coding with the total process of programming. Coding means turning the program solution into a particular programming language. The larger a program gets, the less important this task becomes in relation to analysis and design. So much so that in commercial programming it is the analysts who get the better-paid jobs whilst coding is treated as a lower level task.

Step 4 Test the solution
We shall leave this for discussion later (see Unit 10).

SAQ 1
Write down the steps involved in the bubble sort routine:

```
1000 DEF PROCsort(N)
1010 LOCAL I%, exchange
1020
1030 REPEAT
1040   exchange = FALSE
1050   FOR I% = I + 1 TO N - 1
1060     IF A$(I% + 1) < A$(I%) THEN PROCswap
1070     NEXT
1080   UNTIL NOT exchange
1090 ENDPROC
1100
1110 DEF PROCswap
1120 LOCAL T$
1130 T$ = A$(I%) : A$(I%) = A$(I% + 1) : A$(I% + 1) = T$
1140 exchange = TRUE
1150 ENDPROC
```

Program 5.1

This excercise has, of course, been done back to front since we should write down the design in order to code the solution. But you will probably find it easier to start writing down solutions by first having to describe in general terms programs which you have written – but only as an exercise, not as a principle of programming.

5.2 Top-down design
Let's now illustrate the process of designing a program and introduce top-down design at the same time.

The idea behind top-down design is that you define your program in broad terms, dividing it into parts, each of which can be defined in more detail and divided into parts again. You start with the broad picture and gradually work down to the detail. This is a method of

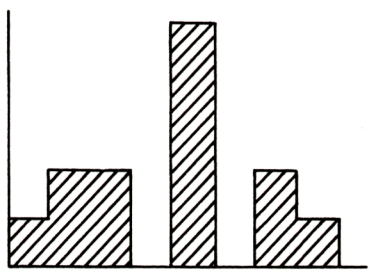
program designing that is made doubly rewarding when using procedures, since they encourage us to define programs initially in broad terms. As an illustration, let's look at the following simple problem.

Example 1

Design a program to accept a date, a set of names of students for up to 50 students and their marks on a test. Marks will be in the range 0 upwards and the total possible marks will also be available. The program should convert the marks to percentages, print out a list of the students in rank order of their marks in the following format:

Student	Percentage
Peter	0
Alice	20
etc.	

and display a histogram of their marks in the following format:



Solution

Step 1 Analyse the problem

Here is a well-defined problem but it isn't clear how we are to receive the data. We shall put it into DATA statements for the purpose of this example but, in practice, the data would have to be fed into the computer in some other way.

Step 2 Design a solution

As with many programming problems a clear statement of what is to be done provides most of what we need for a solution:

Enter the data
Convert the marks to percentages
Sort the students by mark
List the students in sorted order
Calculate the frequencies
Draw a histogram

Now each of these steps could be subdivided into further steps in the description but, in this simple case, they don't require it. In any case, the two most complex tasks (sorting and drawing a histogram) are tasks for which we already have procedures. So let us go on to coding.

Step 3 Code the solution

When we start to code the solution, we notice that each step in the solution is big enough and self-contained enough to be a procedure. So, after the title REMs and other introductory items, we start the program with the main controlling part which will be:

```
100 PROCread_data
110 PROCsort
120 PROClist
130 PROCfrequencies
140 PROChistogram
150
160 END
```

That is what is meant by top-down design. The whole program is there, showing a complete solution to the problem. All we need do now is add two types of material to the program: the preliminary routines and the definitions of the procedures.

Section of program	Content
PRELIMINARY ROUTINES	title description listing of variable names array dimensioning
MAIN CONTROL SECTION	PROCread_data PROCsort PROClist PROCfrequencies PROChistogram
PROCEDURE DEFINITIONS	Detailed definitions

In practice, the BBC Micro requires a change to the main control section because you are not allowed to change Mode inside a procedure. We shall want Mode 5 for the histogram and, say, Mode 7 for the list. So we have to insert the Mode changes into the control section.

```
MODE 7
PROCread_data
PROCsort
```

```
PROClist
PROCfrequencies
MODE 5
PROChistogram
```

SAQ 2

Here now are the first two sections of the program showing the preliminaries and the main control section:

```
 10 REM MARKS LIST AND HISTOGRAM
 20
 30 REM Reads a set of students and marks
 40 REM Ranks the students by mark
 50 REM Prints out a list of students in rank order
 60 REM Draws a histogram of their marks
 70
 80 REM Program variables
 90 REM bands = number of frequency bands
100 REM frequency(I) = frequency of scores in range step *
   (I - 1) to step * I
110 REM limit = maximum number of students allowed by program
120 REM maxscore = total possible marks for any one student
130 REM N = number of students on this run
140 REM name$( ) = names of students
150 REM score( ) = individual scores of students (converted
    to % by program)
160
170 limit = 50
180 DIM name$(limit), score(limit), frequency(limit)
190
200 MODE 7
210
220 PROCread data
230
240 PROCsort
250
260 PROClist
270
280 PROCfrequencies(10)
290
300 MODE 5
310
320 PROChistogram(bands)
330
340 END
```

Program 5.2

Complete the program by adding the procedures. With this method of program design, you only need to add procedures, so all your additions will be below the `END` statement.

5.3 Modular design

The last section demonstrated the use of modular design in building up programs. Why should we adopt this approach? There are four principle reasons:

- ☐ Modular programs are easier to plan and build up in small steps.
- ☐ Modular programs can use ready-made procedures which have previously been thoroughly tested.
- ☐ Modular programs are easy to extend.
- ☐ Modular programs are easier to de-bug.

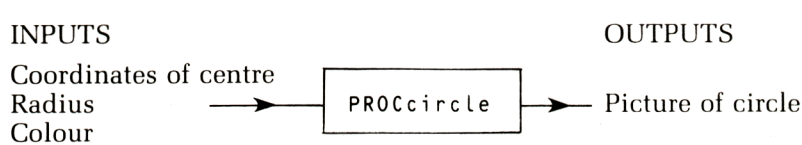
The key to the method lies in procedures, although modular design can be achieved to some extent with subroutines. We shall first explain the principles of modular design using procedures and then discuss what can be done with subroutines.

What we wish to achieve is procedures that are as general as possible and which interfere as little as possible with their host program so that we can build up programs from separate procedures. A perfect procedure for this purpose is one that will not affect any variables outside itself. When that is the case we are able to regard the main program as a host program into which we can drop the procedure without any modification to either the procedure or the host program.

We want to be able to regard our procedures as 'black boxes' i.e. as routines which have inputs and outputs but whose workings need not involve other parts of the program. Consider, for example, a procedure to draw a circle:

```
PROCcircle
```

Its input should be enough to define the circle but no more. Its output will be a circle drawn on the screen. Regarding the procedure as a black box, we can represent it as:



Since there are no output variables, and since the procedure doesn't need to affect global variables, we must ensure that all its variables are local. That will prevent the procedure from interfering with the host program's variables. So the procedure might be:

```
1000 DEF PROCcircle(X%, Y%, radius, colour)
1010 LOCAL step, angle, I%
1020
1030 REM  Set colour, move origin, switch off cursor
1040 GCOL 0, colour
1050 VDU 29, X%; Y%;
1060 VDU 23;8202;0;0;0;
1070
1080 REM Draw circle
1090 MOVE radius * COS(0), radius * SIN(0)
1100 step = 2 * PI / 30 : REM 30 steps makes the polygon
     look like a circle
1110 FOR I% = 0 TO 30
1120   angle = I% * step
1130   DRAW radius * COS(angle), radius * SIN(angle)
1140   NEXT I%
1150
1190 ENDPROC
```

Program 5.3

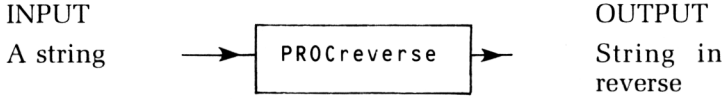
SAQ 3

Program 5.3 is not a 'black box' procedure – it still interferes with the host program in three ways. What are these?

SAQ 4

Write a 'black box' procedure to draw a line box of given size at a given location in a given colour. The parameters to be provided are: width, height, co-ordinates of top left-hand corner and colour.

Program 5.1 is an example of a procedure that does not need to affect global variables or return parameters. Unfortunately BBC BASIC does not allow procedures to return parameters: any output variable must be a global variable. An example of this problem would be a procedure to reverse a string:



Although we can pass the input string as a parameter and use local variables for the activities of the procedure, the output must be a global variable:

```
1000 DEF PROCreverse(A$)
1010 T$ = ""
1030 FOR I% = LEN A$ TO 1 STEP -1
1040   T$ = T$ + MID$(A$, I%, 1)
1050   NEXT I%
1060 ENDPROC
```

Program 5.4

This is unsatisfactory because the procedure can no longer be simply dropped into a program. The host program might already be using T$ as a global variable. In this simple case there is an easy solution - which is to write a user-defined function. Like procedures, defined functions can accept parameters and use local variables but they can also return one parameter. Re-writing Program 5.4 as a defined function we have:

```
1000 DEF FNreverse(A$)
1010 LOCAL T$, I%
1020  T$ = ""
1030 FOR I% = LEN A$ TO 1 STEP -1
1040   T$ = T$ + MID$(A$, I%, 1)
1050   NEXT I%
1060 = T$
```

Program 5.5

which is a 'black box' defined function. We have neatly solved this problem but if we need to return two or more parameters at the same time, neither BBC procedures nor BBC defined functions can help: we have to go back to global variables.

Finally, one more difficult problem in using BBC procedures and functions for modular programming: you cannot (easily) pass an array to a procedure. Suppose you wanted a function to find the highest element in a numeric array. It would be convenient to write something like:

```
1000 DEF FNhighest(A( ),N)
1010 LOCAL I%
1020 max = 0
1030 FOR I% = 0 TO N
1040   IF A(I%) > max THEN max = A(I%)
1050   NEXT I%
1060 = max
```

Program 5.6

Unfortunately, BBC BASIC does not allow a statement as in line 1000. The only reasonably straightforward way round this is to transfer the contents of the array that you would like to pass to the procedure to a temporary array which is actually used in the procedure. This must be

done before entering the procedure so as to keep the coding of the procedure as general as possible. Program 5.7 shows this technique:

```
 10 REM How to 'pass' an array to a function
 20
 30 DIM numbers(10), temp(10)
 40
 50 FOR I% = 0 TO 10
 60   numbers(I%) = RND(100)
 70   NEXT I%
 80
 90 REM Transfer array to be passed to function
100 FOR I% = 0 TO 10
110   temp(I%) = numbers(I%)
120   NEXT I%
130
140
150 PRINT "Highest number in array is " ; FNhighest(10)
160
170 END
180
190 DEF FNhighest(N)
200 LOCAL I%, max,
210 max = 0
220 FOR I% = 1 TO N
230   IF temp(I%) > max THEN max = temp(I%)
240   NEXT I%
250  = max
```

Program 5.7

There is, in fact, a way of passing arrays to procedures which is not part of BBC BASIC but uses a technique described in *The BBC Micro Revealed* (by Jeremy Ruston). The method uses advanced programming techniques and delves deep into the way the BBC Micro stores its variables. (But, be careful, since many of his techniques directly alter memory locations in the BBC Micro and these locations are different for different versions of the BBC Micro's Operating System. Ruston's book was written for OS 0.1).

BASIC without procedures and defined functions

If you use a BASIC without procedures or defined functions you can still use modular programming. However, you only have global variables so every variable in any subroutine is capable of interfering with the host program. You have, therefore, to keep much more careful track of variables and, if necessary, re-name variables before calling a subroutine. For example, here is Program 3.7 as we first wrote it:

```
 10 REM Input check
 20
 30 name_len = 10 : REM Length of name
 40 code_len = 3 : REM Length of area code
 50
 60 PRINT "Enter name (max "; name_len " characters)"
 70 PROCinput(name_len)
 80 name$ = A$
 90
100 PRINT "Enter area code (max "; code_len " characters)"
110 PROCinput(code_len)
120 area$ = A$
130
140 END
150
160 DEF PROCinput(L)
170
180 REPEAT
190   INPUT A$
200   IF LEN(A$) > L THEN PRINT "Too long. Please re-enter."
210   UNTIL LEN(A$) <= L
220
230 ENDPROC
```

and here it is using a subroutine in place of the procedure:

```
 10 REM Input check
 20
 30 name_len = 10 : REM Length of name
 40 code_len = 3 : REM Length of area code
 50
 60 PRINT "Enter name (max "; name_len " characters)"
 65 L = name_len
 70 GOSUB 160
 80 name$ = A$
 90
100 PRINT "Enter area code (max "; code_len " characters)"
105 L = code_len
110 GOSUB 160
120 area$ = A$
130
140 END
150
160 REM Input subroutine
170
180 REPEAT
190   INPUT A$
```

```
200   IF LEN(A$) > L THEN PRINT "Too long. Please re-enter."
210   UNTIL LEN(A$) <= L
220
230 RETURN
```

Because the subroutine does not have parameter passing, we have had to transfer '`name_len`' to '`L`' before the first call of the subroutine and '`code_len`' to '`L`' before the second call. If there were several parameters to pass, this transfer operation would have to be done for all of them.

5.4 Menu driving and command driving

Menu driving

Small programs often do only one thing, e.g. draw a circle or calculate your tax. But as programs get larger, they often do several things, each of which can be regarded as a sub-section of the program. If we are not careful, these sub-sections get built into the program in a fixed manner, giving the user no control over which sections he uses. How much better to offer the user a choice of the segments or even a choice of 'programs' all in one program. This is what happens in menu-driven programs.

A '*menu*' is a list of options which appears on the screen at the start of the program run. For example, a book catalogue program might offer the menu:

1. Search by title
2. Search by number
3. End the run

The user enters the number of the option he wants and the program jumps to that option. Whichever option the user chooses, once he has used that option he usually returns to the menu.

In principle, menu-driving is an extension of top-down design and modular design. The various modules have now become larger, although each main module will be sub-divided into smaller modules until we get down to individual procedures. Pursuing the idea of top-down design, we can write the main control section of the book catalogue program:

```
10 REM Menu driving
20
30 REM Set up menu
40
50 REPEAT
60
70   MODE 7
```

```
 80   PRINT TAB(9, 2) CHR$(141) "BOOK CATALOGUE PROGRAM"
 90   PRINT TAB(9, 3) CHR$(141) "BOOK CATALOGUE PROGRAM"
100   PRINT TAB(0,7) "1 Search by title"
110   PRINT TAB(0,9) "2 Search by number"
120   PRINT TAB(0,11) "3 End"
130
140   REPEAT
150     INPUT TAB(0, 20) "Enter number required:       " option
160     IF option = 1 THEN PROCtitle
170     IF option = 2 THEN PROCnumber
180     IF option = 3 THEN CLS : END
185     UNTIL option >= 1 AND option <= 3
190
200   UNTIL FALSE
```

Program 5.8

This particular menu deals with the problem of invalid options by placing the menu in a REPEAT ... UNTIL ... loop. An invalid option always re-presents the menu as does completion of any valid option. Although we haven't written the rest of Program 5.8, you can get the feel of it even as it stands.

K Type Program 5.8 and try the various menu options.

Menus of menus

In very large programs there may be more options than can be displayed on one menu. In such cases the first menu may lead to a second series of menus as shown in Figure 5.2.

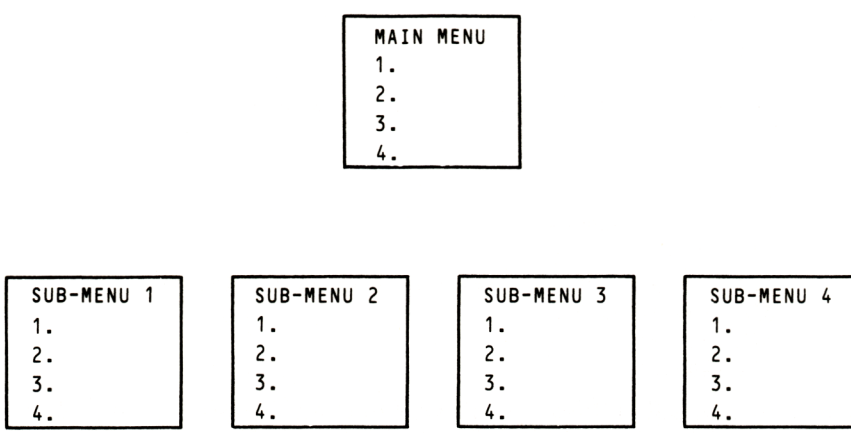


Figure 5.2 A main menu leading to four further menus

SAQ 5

If you have used a general file-creating routine such as Program 5.9 and a general file access routine such as Program 5.10 then you are probably fed up with searching for the 'read' program on your tape after running '`PROCwrite_file`' or vice versa. Link the two routines together into a single menu-driven program:

```
240 DEF PROCwrite_file
250 LOCAL F$, N$, CH
260
270 CLS : PRINT TAB(0, 10) "Enter name for the data
    file. " ;
280 INPUT "" F$
290 CLS
300
310 CH = OPENOUT(F$)
320 REPEAT
330   INPUT "Next item " N$
340   IF N$ <> "ZZZZ" THEN PRINT# CH, N$
350   UNTIL N$ = "ZZZZ"
360 CLOSE# CH
370
380 PRINT F$ " has been saved."
390 PROCreturn
400 ENDPROC
```

Program 5.9

```
430 DEF PROCread_file
440 LOCAL F$, CH
450
460 CLS : PRINT TAB(0,10) "Enter name of the data file " ;
470 INPUT "" F$
480 CLS
490
500 CH = OPENIN(F$)
510 REPEAT
520   IF NOT EOF# CH THEN INPUT# CH,N$ : PRINT N$
530   UNTIL EOF# CH
540 PROCreturn
550 ENDPROC
```

If you are using disk files, you'll need to change lines 520 and 530 to:
Line 520: `IF NOT PTR# CH = EXT# CH THEN INPUT# CH, N$ : PRINT N$`
Line 530: `UNTIL PTR# CH = EXT# CH`

Program 5.10

Command driving

Command driving is a variation on menu driving. The program is structured in the same way as a menu driven program, but the user is allowed to choose his options with commands in English. For example, in the book catalogue program (Program 5.8), the commands permitted would be

```
TITLE
NUMBER
END
```

Program 5.11 shows how these commands can be accepted from the keyboard:

```
 10 REM Menu driving
 20
 30 REM Set up menu
 40
 50 REPEAT
 60
 70   MODE 7
 80   PRINT TAB(9, 2) CHR$(141) "BOOK CATALOGUE PROGRAM"
 90   PRINT TAB(9, 3) CHR$(141) "BOOK CATALOGUE PROGRAM"
100   PRINT TAB(0,7) "TITLE" TAB(12,7) "Search by title"
110   PRINT TAB(0,9) "NUMBER" TAB(12,9) "Search by number"
120   PRINT TAB(0,11) "END" TAB(12,11) "End run"
130   PRINT TAB(0,20) "Enter option required: " ;
140
150   INPUT "" option$
160   option$ = LEFT$(option$,3)
170   IF option$ = "TIT" THEN PROCtitle
180   IF option$ = "NUM" THEN PROCnumber
190   IF option$ = "END" THEN CLS : END
200
210   UNTIL FALSE
```

Program 5.11

After the user has entered his command (line 150), it is shortened to its first three letters and compared to the valid commands for the program. By taking only the first three letters the program is able to accept mis-spelt commands which are nevertheless clear e.g. 'TITLLE' would be interpreted as 'TITLE'.

In simple examples, command driving is probably not worth the extra programming effort when compared to menu driving. Command driving is of value in complex programs where the user needs to give a series of commands such as

```
SEARCH MYFILE SELECT > 1981 SORT PRINT
```

Assignment E

1. Re-design Program 4.11 as follows:

(a) Make it into a procedure that is as self-contained as possible. (Assume that the data is available in data statements. The first data item will be the number of frequencies and the remaining items will be the frequencies).

(b) Show the steps in planning the design using the method described in 5.1.

2. Design a command driven program which displays the menu:

1. Pie chart
2. Histogram
3. End

The options (1) and (2) will use `PROChistogram` (Program 4.6) and the procedure that you have designed in question 1.

Objectives of Unit 5

After studying this unit you should be able to:

- ☐ Plan a program through the three steps of:
  analysis
  design of solution
  coding the solution.
- ☐ Design procedures and user-defined functions in a form suitable for modular program design.
- ☐ Design programs using menu or command driving.

Answers to SAQs

SAQ 1

```
Sort procedure
        Set exchange flag to false
            If next two items in wrong order then swap
            If no more items then end
                If exchange flag false then end

Swap procedure
        Swap the two items
        Set exchange flag to true
```

SAQ 2

```
 10 REM MARKS LIST AND HISTOGRAM
 20
 30 REM Reads a set of students and marks
 40 REM Ranks the students by mark
 50 REM Prints out a list of students in rank order
 60 REM Draws a histogram of their marks
 70
 80 REM Program variables
 90 REM bands = number of frequency bands
100 REM frequency(I) = frequency of scores in range
    step * (I -1) to  step * I
110 REM limit = maximum number of students allowed
    by program
120 REM maxscore = total possible marks for any one student
130 REM N = number of students on this run
140 REM name$( ) =  names of students
150 REM score( ) =  individual scores of  students
    (converted to percentage by program)
160
170 limit = 50
180 DIM name$(limit), score(limit), frequency(limit)
190
200 MODE 7
210
220 PROCread_data
230
240 PROCsort
250
260 PROClist
270
280 PROCfrequencies(10)
290
300 MODE 5
310
320 PROChistogram(bands)
330
340 END
350
360 DEF PROCread_data
370 REM First data item is maximum mark
380 READ maxmark
390
400 REM Read names and marks & change marks to percentages
410 N = 0
```

```
420 REPEAT
430 N = N + 1
440   READ name$(N), score(N)
450   score(N) = (score(N) / maxmark) * 100
460   UNTIL name$(N) = "ZZZZZZ"
470 N = N - 1
480 ENDPROC
490
500 DEF PROCsort
510 LOCAL J, exchange
520
530 REM Sort data by scores
540 REPEAT
550   exchange = FALSE
560   FOR J = 1 TO N - 1
570     IF score(J + 1) < score(J) THEN PROCswap
580     NEXT J
590   UNTIL NOT exchange
610
620 ENDPROC
630
640 DEF PROCswap
650 LOCAL T, T$
660 REM Swap order of scores & names
670 T$ = name$(I) : name$(I) = name$(J) : name$(J) = T$
680 T = score(I) : score(I) = score(J) : score(J) = T
690 exchange = TRUE
700
710 ENDPROC
720
730 DEF PROClist
740 LOCAL I
750 REM Print out list of names in order of scores
760 PRINT "Student", "Percentage"
770
780 FOR I = 1 TO N
790   PRINT name$(I), score(I)
800   NEXT I
810
820 PRINT '' "Press SPACE BAR for histogram."
830 REPEAT : UNTIL GET$ = CHR$32 : REM GET$ introduced
    in Unit 6
840 ENDPROC
850
860 DEF  PROCfrequencies(step)
870 LOCAL frq, counted, I
```

```
 880 REM Count frequency with which each score occurs
 890 FOR I = 1 TO N
 900   counted = FALSE
 910   frq = 0
 920   bands = 0
 930   REPEAT
 940     frq = frq + 1
 950     IF frq > bands THEN bands = frq
 960     IF score(I) <= step * frq THEN frequency(frq) =
         frequency(frq) + 1 : counted = TRUE
 970     UNTIL counted
 980   NEXT I
 990
1000 ENDPROC
1005
1010 DEF PROChistogram(N)
1020 LOCAL I, max, SF, W
1030 VDU 19, 2, 2, 0, 0, 0 : REM Yellow as green
1040 VDU 19, 3, 4, 0, 0, 0 : REM White as blue
1050 VDU 23; 8202; 0; 0; 0;
1060
1070 REM Draw axes
1080 MOVE 0, 0
1090 DRAW 1200, 0
1100 MOVE 0, 0
1110 DRAW 0, 800
1120
1130 REM Find max frequency
1140 max = 0
1150 FOR I = 1 TO N
1160   IF frequency(I) > max THEN max = frequency(I)
1170   NEXT I
1180 IF max = 0 THEN CLS : PRINT "Error. All frequencies
     are zero." : ENDPROC
1190
1200 REM Y axis scale factor
1210 SF = .75 * (1000 / max)
1220
1230 REM Block width
1240 W = 1000 / N
1250
1260 REM Draw blocks
1270 FOR I = 1 TO N
1280   PROCblock(I, W, frequency(I))
1290   NEXT I
1300
```

```
1310 ENDPROC
1320
1330 DEF PROCblock(I, W, frequency(I))
1340
1350 GCOL0, (I MOD 3 +1)
1360 MOVE (I - 1) * W, 0
1370 MOVE (I - 1) * W, frequency(I) * SF
1380 PLOT 85, I * W, 0
1390 PLOT 85, I * W , frequency(I) * SF
1400
1410 ENDPROC
1420
1430 REM Test data
1440 DATA 50
1450 DATA Fred, 30, Jim, 23, Bill, 12, Mary, 25,
     Alice, 10
1460 DATA Ted, 25, Peter, 0, Norman, 30, Maurice, 25,
     Robert, 10
1470 DATA Dora, 12, James, 25, Sheila, 38, ZZZZZZ, 0
```

Program 5.12

SAQ 3

The procedure (a) moves the origin (b) switches off the cursor and (c) changes the graphics foreground colour to `colour`. The following program restores the origin and switches the cursor back on before exiting – the colour change is still left.

```
1000 DEF PROCcircle(X%, Y%, radius, colour)
1010 LOCAL step, angle, I%
1020
1030 REM  Set colour, move origin, switch off cursor
1040 GCOL0, colour
1050 VDU 29, X%; Y%;
1060 VDU 23; 8202; 0; 0; 0;
1070
1080 REM Draw circle
1090 MOVE radius * COS(0), radius * SIN(0)
1100 step = 2 * PI / 30
1110 FOR I% = 0 TO 30
1120   angle = I% * step
1130   DRAW radius * COS(angle), radius * SIN(angle)
1140   NEXT I%
1150
1160 REM Restore origin and cursor
1170 VDU 29,0; 0;
1180 VDU 23; &670A; 0; 0; 0;
1190 ENDPROC
```

Program 5.13

SAQ 4

```
1000 DEF PROCbox(L%, H%, X%, Y%, colour)
1010
1020 REM Set colour, switch off cursor
1030 GCOL 0, colour
1040 VDU 23; 8202; 0; 0; 0;
1050
1060 REM Draw box
1070 MOVE X%, Y%
1080 DRAW X% + L%, Y%
1090 DRAW X% + L%, Y% - H%
1100 DRAW X%, Y% - H%
1110 DRAW X%, Y%
1120
1130 REM Restore cursor
1140 VDU 23; &670A; 0; 0; 0;
1150 ENDPROC
```

Program 5.14

In this procedure, all variables are automatically local since they are all parameters passed to the procedure.

SAQ 5

```
  10 REM File create and access
  20
  30 ON ERROR GOTO 60
  40
  50 REM Menu
  60 REPEAT
  70
  80   MODE 7
  90   PRINT TAB(5, 2) CHR$(141) "FILE CREATE AND ACCESS"
 100   PRINT TAB(5, 3) CHR$(141) "FILE CREATE AND ACCESS"
 110   PRINT TAB(0, 7) "1 Create a file"
 120   PRINT TAB(0, 9) "2 Read a file"
 130   PRINT TAB(0, 11) "3 End run"
 140
 150   PRINT TAB(0, 20) "Key number for option " ;
 160   INPUT "" option
 170   CLS
 180   IF option = 1 THEN PROCwrite_file
 190   IF option = 2 THEN PROCread_file
 200   IF option = 3 THEN CLS : END
 210
 220   UNTIL FALSE
 230
 240 DEF PROCwrite_file
```

```
  250 LOCAL F$, N$, CH
  260
  270 CLS : PRINT TAB(0, 10) "Enter name for the data
      file. " ;
  280 INPUT "" F$
  290 CLS
  300
  310 CH = OPENOUT(F$)
  320 REPEAT
  330   INPUT "Next item " N$
  340   IF N$ <> "ZZZZ" THEN PRINT# CH, N$
  350   UNTIL N$ = "ZZZZ"
  360 CLOSE# CH
  370
  380 PRINT F$ " has been saved."
  390 PROCreturn
  400 ENDPROC
  410
  420
  430 DEF PROCread_file
  440 LOCAL F$, CH
  450
  460 CLS : PRINT TAB(0,10) "Enter name of the data
      file " ;
  470 INPUT "" F$
  480 CLS
  490
  500 CH = OPENIN(F$)
  510 REPEAT
  520   IF NOT EOF# CH THEN INPUT# CH, N$ : PRINT N$
  540   UNTIL EOF# CH
  545 CLOSE# CH
  550 PROCreturn
  560 ENDPROC
  570
  580 DEF PROCreturn
  590 PRINT TAB(0, 22) "Press space bar for menu."
  600 REPEAT : UNTIL GET = 32
  610 ENDPROC
Disk files:
Line 520: IF NOT PTR# CH = EXT# CH THEN INPUT# CH, N$ : PRINT N$
Line 540: UNTIL PTR# CH = EXT# CH
```

Program 5.15

UNIT 6

Data vetting and data entry

6.1 The principles of data vetting

6.2 General input routines

6.3 Other data validation

6.4 Multi-field entry

Assignment F

Objectives of Unit 6

Answers to SAQs

6.1 The principles of data vetting

There are few programs that require no data to be entered from the keyboard or so little data as to be easy to control. Most programs are crucially dependent on the data which is entered by the user. The programs are therefore at the mercy of the user who generally shows an amazing capacity for crashing the program in record time. However, it is possible to design entry systems which are crash-proof. Since such routines are absolutely essential for any program doing real work in an office, factory or school, this unit is devoted to building up to a user-safe entry procedure. We start by considering the problems to be overcome and some of the partial solutions which are available to solve them.

Data types

The two broad categories of data are numeric and string. Superficially it would appear that if we want the user to enter a number, we use a numeric variable and if we want him to enter a string, we use a string variable. Such an approach offers very little protection, as Programs 6.1 and 6.2 show. In Program 6.1, the program accepts non-numeric input but only the digits prior to the first non numeric character are assigned to the variable. The user can input '71Y', which is clearly not a number, yet receive no error message. Program 6.2 accepts both digits and letters, which may or may not be what the program requires. Suppose, however, that the string in Program 6.2 needs to be only alphabetic characters. How could we prevent the acceptance of digits? Using `INPUT` we can only check the entry after the user has pressed Return and there is nothing to inhibit the user from pressing invalid keys. We therefore need to look for a better means of data entry.

```
10 REM Number input test
20
30 REPEAT
40   INPUT "Enter a number " number
50   PRINT number
60   UNTIL FALSE

>RUN
Enter a number 928
       928
Enter a number 71Y
        71
Enter a number R39
         0
```

Program 6.1

```
10 REM String input test
20
30 REPEAT
40   INPUT "Enter a string " string$
50   PRINT string$
60   UNTIL FALSE
```

```
>RUN

Enter a string WORDS AND MORE WORDS
WORDS AND MORE WORDS
Enter a string 1000000 WORDS
1000000 WORDS
```

Program 6.2

K Type and run Programs 6.1 and 6.2 and observe the results of different inputs.

GET

The solution to this problem lies in checking each character as the user presses each key. We don't wait until he has pressed Return by which stage he might have entered up to 255 invalid characters. If a character is invalid, we want to know immediately and be able to reject it.

The BASIC statement that enables us to do this is

```
GET
```

which returns the ASCII number of any key which is pressed. This is demonstrated by Program 6.3.

```
10 REM GET demonstration
20
30 REPEAT
40   X = GET
50   PRINT X
60   UNTIL FALSE
```

Program 6.3

```
>RUN                key pressed
        65              A
        66              B
        62              >
        56              8
       127          Delete
        13          Return
        22          Control and V
```

K Type and run Program 6.3 and investigate the codes of the various

keys. Try some with Caps lock on, some with Shift lock on, some with Control held down, and so on.

As you can see, **GET** doesn't give the character which the key usually gives when using **INPUT** or **PRINT**. To get the character from the ASCII code, we use

```
CHR$(X)
```

Program 6.4 shows how **CHR$(X)** can be used with **GET**.

```
10 REM GET with CHR$ demonstration
20
30 PRINT "X", "CHR$(X)"
40 REPEAT
50   X = GET
60   PRINT ; X, CHR$(X)
70   UNTIL FALSE
```

```
>RUN
X          CHR$(X)
65         A
66         B
62         >
127
13

22
```

Program 6.4

K Type and run Program 6.4. Experiment as in Program 6.3 and find out which keys do not produce printable characters. Note the effects of keys which have ASCII values below 32. These are the screen control characters.

GET$

There is also a command for taking a single string character from the keyboard. The command to do it is

```
GET$
```

which you can use in the form

```
item$ = GET$
```

or, say, as

```
IF GET$ = ...
```

GET$ gives the normal characters as on the keys and is therefore easier to understand. Whether you use **GET** or **GET$** is up to you, however, since whichever you use, you can always convert the results from

ASCII code to CHR$ and vice versa. Most of the routines in this course use GET. If you want to compare GET with GET$, modify Program 6.3 as follows:

Change line 40 to `40 X$ = GET$`.
Change line 50 to `50 PRINT X$`

Returning now to the ASCII numbers obtained from GET, certain characters do not print, e.g. 127 (*delete*), 13 (*return*) and 22 (change Mode). The full effect of all the ASCII codes on the BBC Micro is given in Appendix 1. For the purposes of controlling input, the codes can be broken up into groups:

ASCII codes	Results
1 – 31	Control of the micro's screen and printer
32 – 47	Punctuation marks (including space)
48 – 57	Digits 0 to 9
58 – 63	Punctuation marks
64	@
65 – 90	Capital letters A – Z
91 – 96	Special symbols
97 – 122	Lower-case letters a – z
123 – 126	Special symbols
127	Delete

SAQ 1

What is the connection between the ASCII codes for A – Z and for a – z?

SAQ 2

Write a program that prints the result of a user-defined function for keyboard input that will accept only digits. The accepted characters should be printed one after another on the same line e.g. typing `97A35P` should print `9735`. Pressing Return should end the input routine.

SAQ 3

How would you convert the user-defined function `FNdigits` to return a numeric variable?

SAQ 4

What happens if the user of `FNdigits` enters an incorrect digit and tries to correct his error using Delete?

This shows (as did the provision for 'return') that once you start using GET or GET$ as the input command, you have to decide what is to be done for every possible value of GET or GET$. If you want Delete to be operative, you must arrange for a deletion to take place when `GET = 127`. If you want 'return' to effect 'end of input', then you must arrange for

this when GET = 13. Program 6.5 shows a system using Delete and Return.

```
10 REM Defined function to accept digits only
20
30 PRINT FNdigits
40
50 END
60
70 DEF FNdigits
80 LOCAL X$, N$
90
100 PRINT "Enter a number"
110 REPEAT
120   REPEAT
130     X$ = GET$
140     IF X$ >= "0" AND X$ <= "9" THEN PRINT X$ ; :
        N$ = N$ + X$
150     IF X$  = CHR$(127)  AND LEN(N$)  THEN PRINT  X$ ;  :
        N$ = LEFT$(N$, LEN(N$) - 1)
160     UNTIL X$ = CHR$(13)
170   UNTIL N$ <> ""
180
190 PRINT
200 = VAL N$
```

Program 6.5

6.2 General input routines

SAQ 5

Write a general string input routine as a procedure that will accept any printable character (including Space). Return and Delete should be usable. The procedure should be sent 1 parameter, being the maximum length of input permitted. If the user tries to exceed the maximum length, the procedure should refuse the character and sound a warning note.

Once you have completed SAQ 5, you will have developed a general string input routine that accepts only printable characters and halts when the string reaches the maximum permitted length. When it halts, the Delete and Return keys are the only valid keys. You can use this routine in any program where general string input is to be allowed and where different strings have different maximum lengths.

General numeric input routine

This is not so simple because what constitutes valid numeric input depends on the type of number permitted by the program, i.e. the program may accept:

Positive integers
0
Negative integers
Positive decimals
Negative decimals

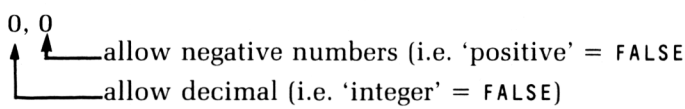
The implications for an input routine are:

Positive integers	Accepts 0, 1, 2 ... 9
Negative integers	As for positive integers but also accepts '-' as the first character
Positive decimals	As for positive integers but also accepts one '.' as a character
Negative decimals	As for positive decimals but also accepts '-' as the first character.

The general numeric input routine needs to be sent two parameters: 'integer' and 'positive' which will each be set to TRUE or FALSE.

Parameter	Values and meanings	ASCII numbers accepted
Integer	-1 = number is to be integer	48 - 57
	0 = number is to be decimal	48 - 57 and 46 (= '.')
Positive	-1 = number is to be positive	
	0 = number is to be positive or negative	43 (= ' + ') and 45 (= '-') the first position only

The permitted numbers for any particular case will come from looking at both parameters, i.e. if a decimal integer (positive or negative) is to be allowed, the parameters sent are



This results in codes 48 to 57, 46, 43 and 45 all being accepted.

The design of the routine uses two defined functions as follows.

FNenter

Set 'point' to FALSE

Enter a character

If the character is Delete then delete last character

If the character is valid, then add it to the number

If the character is Return then end

FNcheckascii

If decimals are allowed and the character is '.' then accept the character and set point to TRUE

If negative numbers are allowed and this is the first character and it is '+' or '-' then accept it

If the character is >= 0 and <= 9 then accept it

If the character was not accepted then make a sound

Program 6.6 shows the coding for this design.

```
200 DEF FNenter(integer, positive)
210 LOCAL entry$, X, last$, point
220 entry$=""
230
240 REPEAT
250   point = FALSE
260   REPEAT
270     X = GET
280       IF  X =  127 AND  LEN(entry$) THEN  PRINT CHR$(X)  ;
          :  last$ = RIGHT$(entry$, 1) : entry$ = LEFT$(entry$,
          LEN(entry$) - 1) :  IF last$  = "." THEN point = FALSE
290     IF X <> 13 AND X <> 127 THEN IF FNcheckascii THEN
        entry$ =  entry$ + CHR$(X) : PRINT CHR$(X) ;
300     UNTIL X = 13
310   UNTIL entry$ <> "-" AND entry$ <> "+" AND entry$ <>  ""
      AND  entry$ <> "."
320 PRINT
330  = VAL(entry$)
340
350 DEF FNcheckascii
360 LOCAL valid
370 valid = FALSE
380
390 IF NOT integer AND point = FALSE  AND CHR$(X) = "." THEN
    valid =  TRUE : point = TRUE
400 IF NOT positive AND LEN(entry$) = 0 AND (CHR$(X) = "+" OR
    CHR$(X) = "-") THEN valid = TRUE
410 IF CHR$(X) >=  "0" AND CHR$(X) <= "9" THEN valid = TRUE
420 IF NOT valid THEN VDU 7
430 = valid
```

Program 6.6

SAQ 6
Write a routine to test Program 6.6 for positive integer entry.

6.3 Other data validation

Our efforts in this unit so far have largely been to ensure that the right type of data is entered. The only additional feature that we have covered is to ensure that in the case of string data, the maximum string length permitted by the main program is not exceeded. A given program may, however, have a number of other constraints on the data, e.g.

Only 2 decimal places (e.g. for money)
Dates between 1 Jan 1970 and 31 Dec 1989
Ages between 21 and 65
Record numbers between 1 and 1000
etc.

These require a further stage of validation on top of the procedure to ensure the right type of input. Some of this validation is very complex. For example, a system for detecting that 290281 and 310982 are invalid dates requires a lot of programming. However, the principle of vetting after the input procedure can be demonstrated with a very simple example.

Example 1
A storehouse cuts steel bars to any length between 1m and 10m. Write an input routine which rejects lengths outside these limits and which only accepts valid numeric characters from the keyboard.

Solution
There are two approaches to solving this problem. One is to write a specific routine to do the job. The other is to use `FNenter` with suitable extra statements. The first approach results in the shorter program but takes a longer time to do. The second approach is very easy and quick to write:

```
10 REM Data range validation
20
30 PRINT "Enter length required (1-10m)."
40 REPEAT
50   length = FNenter(0,-1)
60   IF length >= 1 AND length <= 10 THEN ok = -1 ELSE ok = 0
70   IF NOT ok THEN PRINT ' "Out of stock range.
     Please re-enter."
80   UNTIL ok
90 END
```

Program 6.7

Use of INSTR

The statement 'INSTR' can be very useful in vetting data.

INSTR is used to find the starting position of one string in another. If the string we are looking for is not there, it returns the value zero e.g.

```
INSTR("SEARCH", "EAR")
```

would return the value 2 since "EAR" is in "SEARCH" and starts at position 2.

```
INSTR("SEARCH", "EAT")
```

would return the value 0 since "EAT" is not in "SEARCH".

Example 2

A program is to accept any one of the codes A, B, C, D or E in upper- or lower-case from the keyboard. Write an input routine for this.

Solution

You could solve this using OR but the result would be a very long test line. With INSTR the program is very neat:

```
10 REM Use of INSTR in entry validation
20
30 REPEAT
40
50   REPEAT
60     INPUT "Enter next code " R$
70     T = INSTR("ABCDE", CHR$(ASC R$ AND &DF))
80     UNTIL LEN R$ = 1 AND T <> 0
90   PRINT R$
100
110  UNTIL FALSE
```

Program 6.8

K Type and run Program 6.8.

INSTR is very useful in command driven programs (see Unit 5) where we need to check an input string against a large number of valid commands. This is demonstrated in the next example.

Example 3

A program is to accept one of the commands START, STOP, SORT or PRINT in upper or lower case. Write an input routine for this.

Solution

Following the last example, you might think that the solution is:

```
10 REM Use of INSTR in command driving
20
30 REPEAT
40   REPEAT
```

```
50      INPUT "Enter next command " command$
60      UNTIL INSTR("STARTstartSTOPstopSORTsortPRINTprint",
        command$)
70    PRINT command$
80
90    UNTIL FALSE
```

Program 6.9

SAQ 7

K Type and run Program 6.9. It appears to work but has a bug. What is it? Re-write the program to solve this problem.

Note
Early versions of BBC BASIC have a bug in `INSTR` which causes trouble if the string you are looking for is longer than the string you are looking in. If a search of this type could occur in your program, you will need to insert a trap to prevent it occuring. We could amend Program 6.8, line 70 to:

```
70 T = 0 : IF LEN R$ < 11 THEN T = INSTR("ABCDEabcde", R$)
```

6.4 Multi-field entry

What we have looked at so far are routines for programs with single item entries, that is, where only one piece of data is requested at a time. We now come to a major programming problem which we touched on in Unit 3 – multi-field entry. In multi-field entry, the screen shows what is usually a record made up of several different fields. It is not enough for the user to enter a record field by field: he must consider whether the total entry is correct before allowing it to be passed to the next section of the program. To make matters more complicated, he needs to be able to edit the entry on the screen. This is a big enough problem to require the setting out of the stages of its solution using the analysis and design approach described in Unit 5.

Analysis
There are many ways to analyse a problem. One very useful method in program design is to describe what the program is to do. Where the program includes activities on the screen, the description should include drawings of the proposed screen display, although design of the precise layout is part of the design stage. We can describe the multi-field input routine as a four-stage program:

Stage 1 Request of record:

Which record do you want? 25

Stage 2 Record displayed:

Record number 25

```
Name              : Mrs A L Greenwood :
Address           : Hill Top Farm             :
Address           : Southlands                :
Address           : Northshire                :
Tel.No.           : Southlands 1234   :
```

Stage 3 Edit

The user must now be able to edit the record but without being able to exceed the field lengths (denoted by ':').

Stage 4 Accept

The user must be able to tell the computer to accept the entry as it is on the screen.

User operations

The user must be able to perform all of the following operations:

- ☐ Select the field he wants.
- ☐ Add characters to the field.
- ☐ Delete characters from the field (with the gaps being closed up automatically).
- ☐ Insert characters into the middle of a field.
- ☐ Tab left and right across a field.
- ☐ Tell the computer to accept his entry.

The following keys are to execute these functions:

Function	Key	ASCII number of key
Tab right	→(1)	137
Tab left	←(1)	136
Move to field above	↑(1)(2)	139
Move to field down	↓(1)(2)	138
Accept field	Return(3)	13
Start insert mode	CTRL _	31
End insert mode	CTRL I	28
Delete character	Delete	127
Accept screen	CTRL]	29

Notes

(1) These keys do not normally give these ASCII numbers. To get them to do so, you have to put `*FX 4, 1` in your program which disables their editing function. (`*FX 4, 0` restores their editing function).

(2) This key will only be valid at the start of a field.

This looks more complex than it is. As we build up the program, function by function, it will prove very easy to use. Indeed we cannot over-emphasise that what we are now embarking on is a difficult piece of programming designed to make multi-field entry very easy for the user.

Design

The steps in the program will be:

Display the field titles and field boundaries
Display the current entry for the record

Enter character
If character is 9 and not at end of field then tab right
If character is 136 and not at start of field then tab left
If character is 29 and at start of a field then exit
If character is 127 then *delete previous character*
If character is 31 then *start inserting*
If character is >31 and <127 then *add to the field*
If character is 28 then *end inserting*
If character is 13 then accept field up to cursor and *move to next field*
If character is 138 and at start of a field then *move to next field*

The italicised actions are ones which we anticipate will prove complex enough to require separate procedures. So, at this stage, the design looks like Figure 6.1.

Coding and testing

The standard order of the steps for constructing a program would suggest that we now code Figure 6.1 and then finish off by testing. That is fine for short programs but not for long ones. What we now do is alternate between coding and testing. We code one procedure, test it, code the next, test that, and so on. The order in which we select the procedures to code is the order dictated by top-down design. We therefore start with the highest level of procedure and gradually work down to the lowest.

We shall now proceed to build the program, procedure by procedure. It will be a procedure (or set of procedures) starting at line 20000 and so should be well out of the way of any main program.

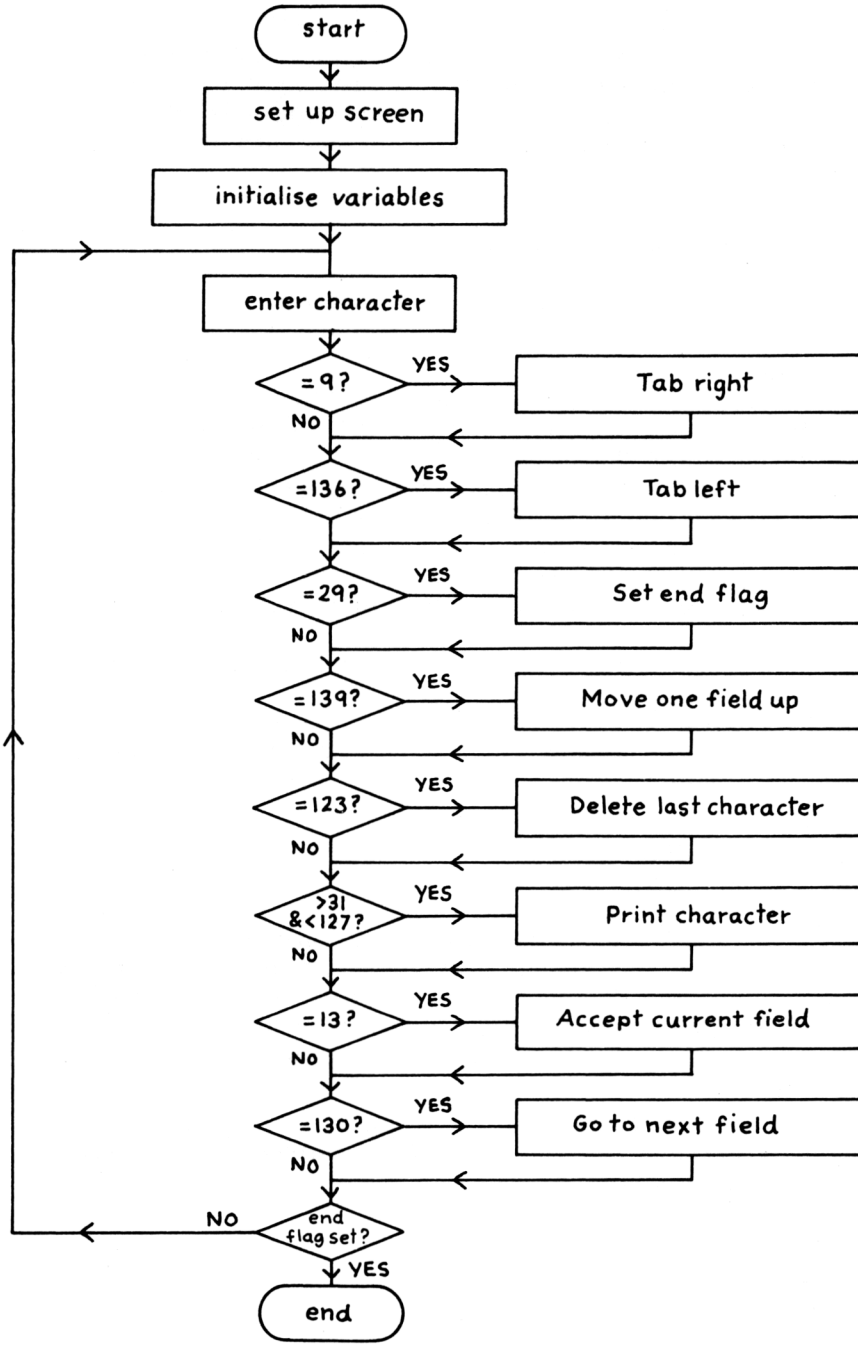


Figure 6.1 Structure of multiple field input routine

1 The test routine

To develop and test the entry procedure, we need a section of test program. This tells the procedure how many fields we want, what their titles are and what their lengths are. It also gives the screen a title and sets up the array dimensions. Here is the test program for 5 fields. Note that we have inserted dummy data into the fields `F$(1)` to `F$(5)` as if they were a record called off a file.

```
 10 REM Test routine for PROCinput
 20
 30 DIM H1(5), H2(5), L(5), F$(5), T$(5), V1(5), V2(5)
 40
 50 REM Field titles
 60 T$(1) = "Name" : T$(2) = "Address" : T$(3) = T$(2) : T$(4) =
    T$(2) : T$(5) = "Tel. No."
 70
 80 REM Data for positions of fields on screen
 90 FOR 1 = 1 TO 5
100   H1(I) = 1
110   H2(I) = 12
120   V1(I) = 4 + I
130   V2(I) = 4 + I
140   NEXT I
150
160 REM Test data
170 F$(1) = "Mrs A L Greenwood" : F$(2) = "Hill Top
    Farm" : F$(3) = "Southlands"
180 F$(4) = "Northshire" : F$(5) = "Southlands 1234"
190
200 REM Field lengths
210 L(1) = 20 : L(2) = 25 : L(3) = 25 : L(4) = 25 : L(5) = 15
220
230 REM Call the input procedure
240 PROCinput(5, 1, "Members' address file")
250
260 REM Check that the procedure worked
270 FOR I = 1 TO 5
280   PRINT F$(I)
290   NEXT I
300
310 END
```

Program 6.10a

(We have used multi-statement lines for test data and field data since this is essentially trivial and does not require a more extensive layout).

2 Set up the main procedure

The procedure starts by:

20010	Declare which variables are local. (We will explain each variable as we use it).
20030 to 40	Clear the screen. Disable the cursor editing keys which we wish to use as control keys.
20060 to 90	Print the fields and field limits on the screen. These are defined by the data in the arrays H1( ), H2( ), V1( ), V2( ) and L( ). The position on the screen of the heading for field I is defined by (H1(I), V1(I)). The position on the screen of the start of the entry for field I is defined by (H2(I), V2(I)). The maximum length for an entry in field I is held in L(I).
20100	Print the heading.
20140 to 20210	Set the initial values of variables not already declared. Note the statement '`WIDTH 0`' (its default value). This is needed since, on entry to the procedure, `WIDTH` may be already set to some value other than zero. If, say, `WIDTH` were set at 5 then as soon as 5 characters had been typed, a carriage return would be executed by the operating system. With `WIDTH 0`, carriage returns are only executed by the operating system when we try to type over the end of full screen lines.

```
20000 DEF PROCinput(fields, start, head$)
20010 LOCAL F$, X, I, J, count, chars, max, insert, T$,
      end, field, valid
20020
20030 CLS
20040 *FX 4, 1
20050 REM Set up screen
20060 FOR I = 1 TO fields
20070      PRINT  TAB(H1(I),  V1(I))  ;  T$(I)  TAB(H2(I)-1,
V2(I))  ":" F$(I) TAB(H2(I)-1, V2(I)) ;
20080   PRINT STRING$(L(I), CHR$(9)) CHR$(9) ":"
20090   NEXT I
20100 PRINT TAB(20 - (LEN(head$) DIV 2) , 2) ; head$
20110 PRINT TAB(12, 21) ; "CTRL ] to enter."
20120
20130 REM Initialise variables
20140 chars = 0
```

```
20150 count = 0
20160 max = 1
20170 insert = FALSE
20180 T$ = ""
20190 end = FALSE
20200 field = start
20210 WIDTH 0
20220
20230 REM Accept characters from screen & decide on
      action to take
20240 REPEAT
20250
20270   X = GET
20400
20410   UNTIL end
20420
20430 REM Leave input procedure : new record is in F$( )
20440 CLS
20450 *FX 4,0
20460 ENDPROC
```

Program 6.10b

K Type and run Programs 6.10a and 6.10b. You should get a clear display of the test data and fields but as yet you cannot alter any of the data. At this stage you are able to detect any errors in the screen display routine. Are the headings where they should be? Are the field markers in the correct positions? Is the record correctly displayed? All these things should be checked now, taking advantage of the fact that top-down programming allows us to work errors out as we go along.

3 Typing into a field

To type into a field, we need to move the cursor to the start of the field. The number of the field in use is noted by the variable `field` which is set to `start` by line 20200. `start` is one of the parameters sent to the procedure and is the number of the first field which the user can alter. Normally `start` will be 1 but if the first field was a file record number which must not be altered, then we could make `start` equal to 2. Line 20140 sets the counter `chars` to zero. `chars` counts the number of valid characters input so far into the current field.

Add

```
20260 IF chars = 0 THEN PRINT TAB(H2(field), V2(field)) ;
```

to your program to set the cursor to the start position of the current field.

Every time that a character is input, we set the flag `valid` to `FALSE`. If, later in the program, we find the character was a valid character which will be added to the current field or was a valid control character, we set 'valid' to `TRUE`. If the character proves invalid, we sound a warning beep. Add line 20390 to do this:

```
20390 IF NOT valid THEN VDU 7
```

K Try Program 6.10 with these extra lines. You should find that the cursor moves to the start of the first field and that whichever key you press, you get a warning beep. That is as it should be – as yet we haven't told the program which characters are valid.

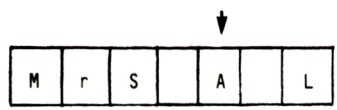
Accepting valid print characters

We want to be able to accept characters between `CHR$ 32` (space) and `CHR$ 126` (~) and print them into the fields. Add the line:

```
20350 IF CHR$(X) >= " " AND CHR$(X) <= "~" THEN chars =  chars +
1 :  valid = TRUE : PROCcharacter
```

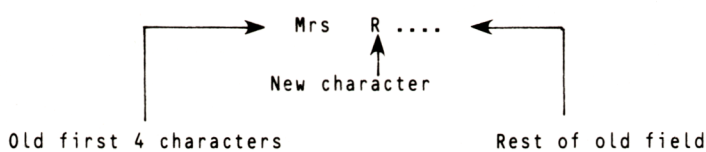
Note '`valid = TRUE`'. We have detected a valid character from the keyboard so we change `valid` from `FALSE` to `TRUE` to prevent the warning beep at the end of the loop (line 20390).

What shall we do with the character? Well, suppose it is entered at the fifth position in field 1:



and that the user is trying to change the initial to 'R'.

At the moment, `F$(1) = "Mrs A L Greenwood"`. We want to change `F$(1)` to



Program 10c does this:

```
20480 DEF PROCcharacter
20490 REM Adds new character to current field
20500 IF insert THEN count = count + 1
20510 F$ = F$(field)
20520 len = LEN F$
20530 F$(field) = LEFT$(F$, chars - 1) + CHR$(X)
20540 PRINT CHR$(X) ;
```

```
20550 IF len > chars THEN F$(field) = F$(field)
      + RIGHT$(F$,len - chars)
20560 IF chars = L(field) THEN PROCaccept_field
20570 ENDPROC
```

Program 6.10c

K Add Program 6.10c to your routine. Run the program and type *over* the first field. At the moment, there is no proper way out of this, so to check what has happened, press ESCAPE and type

```
PRINT F$(1)
```

You should find that F$(1) starts with the new characters you typed in and then continues with the rest of 'Mrs A L Greenwood'.

Line 20560 detects whether or not the addition of the latest character has filled the current field. If it has, control is switched to PROCaccept_field. This moves the cursor to the start of the next field.

SAQ 8

Write PROCaccept_field. It should:

- ☐ Blank out the current field from the cursor position to the end of the field.
- ☐ Make the current field take the string entered so far.
- ☐ Remove any trailing blank spaces from the accepted field.
- ☐ Move on to the next field.

4 Tabbing

Tabbing left and right is straightforward.

We permit tabbing forward only up to the end of the field:

```
20290 IF X = 137 AND chars <= LEN F$(field) - 1 THEN chars =
chars + 1 : valid = TRUE : VDU 9
```

(VDU 9 moves the cursor forward by one position).

(If you also want to allow the user to use the 'TAB' key (CHR$9) for tabbing, then write IF (X = 137 OR X = 9) ...)

We permit tabbing backwards only to the start of a field:

```
20300 IF X = 136 AND chars > 0 THEN valid = TRUE : chars = chars
- 1 : VDU 8
```

(VDU 8 backspaces the cursor by one position).

K Add lines 20290 and 20300 and test the tabbing. Check for the 'beep' when illegal tabbing is attempted.

5 Deleting

At the moment, the delete key is out of action so the user cannot delete a character except by typing over it. We are now going to arrange for

Delete to remove the character at the current cursor position and to close up the string to the right of the gap one character leftwards. Delete is not allowed beyond the right-hand end of the current entry in the current field.

The line to activate Delete is

```
20330 IF X = 127 AND chars <= LEN F$(field) - 1 THEN valid =
TRUE : PROCdelete
```

SAQ 9

Write `PROCdelete` and test it.

6 Changing fields

You have seen how typing over the end of a field (line 20560) automatically moves the cursor onto the next field. There are two other ways in which a field change can take place:

- ☐ Pressing 'return' in the middle of a field accepts the field up to the cursor position and then moves to the next field.

If the cursor is at the start of a field:

- ☐ Pressing '↓' moves to the next field down.
- ☐ Pressing '↑' moves to the next field up.

These facilities are provided by:

```
20370 IF X = 13 AND chars > 0 THEN valid = TRUE : PROCaccept_field
20380 IF X = 138 AND chars = 0 THEN valid = TRUE : PROCnext_field
20320 IF X = 139 AND chars = 0 AND field > start THEN valid =
      TRUE :  field = field - 1
```

The procedure to move to the next field is quite simple:

```
21120 DEF PROCnext_field
21130 IF field = fields THEN field = start ELSE field = field + 1
21140 chars = 0
21150 ENDPROC
```

Ⓚ Type these additions and test changing fields.

7 Accepting the screen

When the user is satisfied with the screen, he can ask the computer to accept the screen by typing '`control ]`' i.e. by holding down '`CTRL`' and pressing '`]`'. (The cursor must be at the start of any field). This is done with:

```
20310 IF X = 29 AND chars = 0 THEN valid = TRUE : end = TRUE
```

8 Inserting

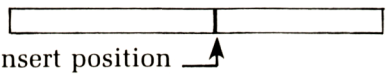
The user is allowed to insert characters into the middle of a field provided (a) the field is not full and (b) a previous insert operation is

not still in operation. (a) can be checked from existing variables. (b) requires the introduction of a new flag, `insert`. If an insert operation is in progress, then `insert = TRUE`, otherwise `insert = FALSE`. To activate the insert routine we use the line:

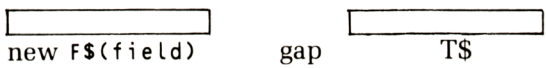
```
20340 IF X = 31 AND chars <= LENF$(field) -1 AND LEN F$(field) <
L(field) AND insert = FALSE THEN valid = TRUE : PROCinsert_open
```

Insert works as follows

(a) `F$(field) =`



(b) Split `F$(field)` open at the insert position.
Store the right part as `T$`.
Make `F$(field)` equal to the left part.
Put up to 20 blank spaces in the gap.



'max' notes the gap size at the start
'count' notes how many characters have been put in the gap (using `PROCcharacter` as usual).

(c) The insert mode continues until either the gap is filled or the user closes it by typing Control. This is activated by the line:

```
20360 IF X = 28 AND chars > LEN F$(field) -1 OR count = max THEN
valid = TRUE : PROCinsert_close
```

`chars > LEN F$(field) -1` ensures that the user cannot close the insert gap whilst he is tabbed out of it.

The procedure is as shown in Program 6.11.

```
20820 DEF PROCinsert_open
20830 insert = TRUE
20840 F$ = F$(field)
20850 len = LEN F$
20860 T$ = ""
20870 max = L(field) - LEN F$(field)
20880 IF max > 20 THEN max = 20
20890 F$(field) = LEFT$(F$, chars)
20900 T$ = RIGHT$(F$, len - chars)
20910 PRINT STRING$(max, "-") ;
20920 PRINT T$ ;
20930 PRINT STRING$(max + len -chars, CHR$(8)) ;
20940 ENDPROC
```

Program 6.10d

Comments

20830	Set the 'insert mode on' flag.
20870 to 20880	Note how the appropriate value of max (the size of the gap) is calculated.
20890 to 20900	Split the old F$(field) into the part left of the gap which becomes F$(field) and the part to the right of the gap which is held in `T$` until the end of the insertion.
20910	Print the gap on the screen.

One more modification is needed which is to count the number of characters typed into the gap. This is done by a modification to `PROCcharacter` where we add the line

```
20500 IF insert THEN count = count + 1
```

K Add Program 6.11 to Program 6.10. Run the program and check that you can open an insert gap and fill it. (You can't close it yet).

(Another approach to the inserting problem is to provide a key that puts the screen into inserting mode. Then any keys typed are inserted rather than over-typed. This approach is not very user-friendly since it is never absolutely clear that insert mode is operating. The gap that we use makes inserting explicit).

SAQ 10

Try writing `PROCinsert_close`. See if you can get it to close up the gaps made by `PROCinsert`.

Summary

```
20000 DEF PROCinput(fields, start, head$)
20010 LOCAL F$, X, I, J, count, chars, max, insert,
      T$, end, field, valid
20020
20030 CLS
20040 *FX 4, 1
20050 REM Set up screen
20060 FOR I = 1 TO fields
20070      PRINT  TAB(H1(I),  V1(I))  ;  T$(I)  TAB(H2(I)-1,
           V2(I))  ":" F$(I) TAB(H2(I)-1, V2(I)) ;
20080   PRINT STRING$(L(I), CHR$(9)) CHR$(9) ":"
20090   NEXT I
20100 PRINT TAB(20 - (LEN(head$) DIV 2) , 2) ; head$
20110 PRINT TAB(12, 21) ; "CTRL ] to enter."
20120
20130 REM Initialise variables
20140 chars = 0
```

```
20150 count = 0
20160 max = 1
20170 insert = FALSE
20180 T$ = ""
20190 end = FALSE
20200 field = start
20210 WIDTH 0
20220
20230 REM Accept characters from screen & decide on
      action to take
20240 REPEAT
20250
20260   IF chars = 0 THEN PRINT TAB(H2(field), V2(field)) ;
20270   X = GET
20280   valid = FALSE
20290   IF (X=137 OR X=9) AND chars <= LEN F$(field)-1 THEN chars
        = chars  + 1 : valid = TRUE : VDU 9
20300   IF X=136 AND chars > 0 THEN valid = TRUE :
        chars = chars -  1 :  VDU 8
20310   IF X=29 AND chars = 0 THEN valid = TRUE : end = TRUE
20320   IF X=139 AND chars = 0 AND field > start THEN valid =
        TRUE :  field = field - 1
20330   IF X=127 AND chars <= LEN F$(field)-1 THEN valid  =
        TRUE  : PROCdelete
20350   IF CHR$(X) >= " " AND CHR$(X) <= "~" THEN chars = chars +
        1 :  valid = TRUE : PROCcharacter
20370   IF X=13 AND chars > 0 THEN valid = TRUE : PROCaccept_field
20380   IF X=138 AND chars = 0 THEN valid = TRUE : PROCnext_field
20390   IF NOT valid  THEN VDU 7
20400
20410   UNTIL end
20420
20430 REM Leave input procedure : new record is in F$( )
20440 CLS
20450 *FX 4,0
20460 ENDPROC
20470
20480 DEF PROCcharacter
20490 REM Adds new character to current field
20500 IF insert THEN count = count + 1
20510 F$ = F$(field)
20520 len = LEN F$
20530 F$(field) = LEFT$(F$, chars - 1) + CHR$(X)
20540 PRINT CHR$(X) ;
20550 IF len > chars THEN F$(field) = F$(field) +
      RIGHT$(F$,len - chars)
```

```
20560 IF chars = L(field) THEN PROCaccept_field
20570 ENDPROC
20580
20590 DEF PROCaccept_field
20600 REM Puts contents of current field  from start  to cursor
      position into F$(field)
20610 REM Blank out field to right of cursor
20620 PRINT STRING$(L(field) - chars, " ") ;
20630 F$(field) = LEFT$(F$(field), chars)
20640
20650 REM Strip off blank spaces on right of field entry
20660 stripped = FALSE
20670 REPEAT
20680   IF RIGHT$(F$(field), 1)  = "  "   THEN F$(field)  =
        LEFT$(F$(field), LEN(F$(field)) - 1) ELSE stripped = TRUE
20690   UNTIL stripped
20700 PROCnext_field
20710
20720 ENDPROC
20730
20740 DEF PROCdelete
20750 F$ = F$(field)
20760 len = LEN F$
20770 F$(field) = LEFT$(F$, chars) + RIGHT$(F$, len - chars - 1)
20780 PRINT RIGHT$(F$, len - chars - 1) " " ;
20790 PRINT STRING$(len - chars, CHR$(8)) ;
20800 ENDPROC
20810
20820 DEF PROCinsert_open
20830 insert = TRUE
20840 F$ = F$(field)
20850 len = LEN F$
20860 T$ = ""
20870 max = L(field) - LEN F$(field)
20880 IF max > 20 THEN max = 20
20890 F$(field) = LEFT$(F$, chars)
20900 T$ = RIGHT$(F$, len - chars)
20910 PRINT STRING$(max, "-") ;
20920 PRINT T$ ;
20930 PRINT STRING$(max + len - chars, CHR$(8)) ;
20940 ENDPROC
20950
20960 DEF PROCinsert_close
20970 REM Reconstitute field from left part in F$(field) and
      saved right part in T$
20980 F$(field) = LEFT$(F$(field), chars + 1)
```

```
20990 len = LEN F$(field)
21000 F$(field) = F$(field) + T$ : chars = LEN F$(field)
21010
21020 REM Print new field
21030 PRINT TAB(H2(field), V2(field)) F$(field) ;
21040 REM Blank out field to right of end of field
21050 PRINT STRING$(L(field) - chars, " ") ;
21060
21070 REM Move cursor back to inserting position
21080 PRINT STRING$(L(field) - len, CHR$(8)) ;
21090 chars = len
21110 insert = FALSE
21120 T = 0
21130 max = 1
21140 count = 0
21150 T$ = ""
21160 ENDPROC
21700
21180 DEF PROCnext_field
21190 IF field = fields THEN field = start ELSE field =
      field + 1
21200 chars = 0
21210 ENDPROC
```

Program 6.10

SAQ 11

The point of `PROCinput` is to be able to use it to create or amend records. Write a program to hold records of names and addresses (3 lines) and telephone numbers in arrays, `name$`, `address1$`, `address2$`, `address3$` and `tel$`. Use `PROCinput` to allow the user to access any one of these records and amend it or create a new record at that position. Make the maximum number of records equal to 50.

Assignment F

1. Write a string input procedure that accepts any of the following characters into the string:

    Capital letters
    Lower case letters
    ',', '-' and '.'

The maximum string length accepted should be a parameter passed to the procedure. Delete should operate normally and Return should end the input. Pressing any invalid key should sound a warning beep.

2. Write a numeric input procedure for entering sterling currency figures, i.e. figures in the form `xx` for round pounds, or `xx.xx` for pounds and pence or `.xx` for pence only. Any incorrect entries should

produce the message 'Invalid sterling format. Please re-enter.' Pressing any invalid key should sound a warning beep.

(The user should not enter '£' or 'p' but you may wish to show these in the result).

Objectives of Unit 6

After studying this unit you should be able to:

- ☐ Use GET and CHR$ to take single characters from the keyboard.
- ☐ Create input routines to accept string characters of a type defined in a program specification.
- ☐ Create input routines to accept numbers of a pre-defined type.
- ☐ Write single data-vetting routines.
- ☐ Use the multi-field entry procedure in data-processing programs and modify it to suit special needs.

Answers to SAQs

SAQ 1

The ASCII number for a lower-case letter is 32 higher than for the same letter in upper-case. This allows us to use the bit-wise AND as a mask to convert all letters typed into upper case. ASCII values in the range 87 to 122 (a to z) are converted to numbers in the range 65 to 90 (A to Z). This is demonstrated by the following program. Try typing in capitals and lower case and watch the result.

```
10 REPEAT
20   X = GET
30   IF X = 13 THEN PRINT
40   PRINT CHR$(X AND &DF) ;
50   UNTIL X = 13
```

(If you have not met the bitwise use of AND, then we suggest that you skip this answer and the solution to exercise 2).

SAQ 2

```
10 REM Function to accept only digits
20
30 PRINT FNdigits
40
50 END
60
70 DEF FNdigits
80 LOCAL X$, N$
90
```

```
100 PRINT "Enter a number:"
110 REPEAT
120   X$ = GET$
130   IF X$ >= "0" AND X$ <= "9" THEN PRINT X$ ; :
      N$ = N$ + X$
140   UNTIL X$ = CHR$(13)
150
160 PRINT
170 = N$
```

Program 6.12

```
>RUN
Enter a number
9735                - The user entered 97A35P followed by 'Return'.
```

SAQ 3

Change `170 = N$` to `170 = VAL(N$)`

SAQ 4

Nothing. Delete (= ASCII 127) is rejected by the test at line 130.

SAQ 5

```
 10 REM Defined function for valid string entry
 20
 30 N = 5
 40 PRINT "Enter up to " ; N " characters"
 50 PRINT FNstring_enter(N)
 60
 70 END
 80
 90 DEF FNstring_enter(max)
100 LOCAL entry$, K$, full
110 entry$ = ""
120 full = FALSE
130
140 REPEAT
150   K$ = GET$
155   REM Check for valid character
160   IF K$ >= " " AND K$ <= "~" AND NOT full THEN PRINT K$
      ; :  entry$ = entry$ + K$
165   REM Check for delete
170     IF  K$ =  CHR$(127) AND LEN(entry$) THEN PRINT K$ ;
        :  entry$ = LEFT$(entry$, LEN(entry$) - 1)
175   REM Beep if invalid character
```

```
180    IF full AND K$ <> CHR$(13) AND K$ <> CHR$(127) THEN VDU 7
190    IF LEN(entry$) = max THEN full = TRUE ELSE full = FALSE
200    UNTIL K$ = CHR$(13)
210
220 PRINT
230 = entry$
```

Program 6.13

SAQ 6

```
 10 REM Test FNenter on positive integers
 20
 30 PRINT "Enter a positive integer: " TAB(35) ;
 35 integer = TRUE : positive = TRUE
 40 number = FNenter(integer, positive)
 50 PRINT ' "Check number returned by FNenter: "
    TAB(35) ; number
 60 END
```

Program 6.14

(We've chosen to tab to 35 in order to improve the screen layout).

SAQ 7

The bug is that the program accepts partial commands such as `STA` or `PRI` or, worse still, `S`. The solution to this is to pad the command entered with a heading and trailing space:

```
command$ =  " "  +  command$ +  " "
```

and to pad the reference string to separate the words:

```
START start STOP ... etc.
```

The bug-free routine is then:

```
 10 REM Use of INSTR in command driving
 20 REM improved version
 30
 40 REPEAT
 50
 60   REPEAT
 70     INPUT "Enter next command " command$
 80     command$ = " " + command$ + " "
 90     T = INSTR(" START start STOP stop SORT sort PRINT
        print ", command$)
100     UNTIL T <> 0
110   PRINT command$
120
130   UNTIL FALSE
```

Program 6.15

This still doesn't accept entries such as 'Start'. We can overcome this by using masking. See SAQ 1 answer, in which case lines 70 and 80 of Program 6.15 would be replaced by

```
PRINT "Enter next command " ;
command$ = ""
REPEAT
  ascii = GET
  letter$ = CHR$(ascii AND &DF)
  IF ascii <> 13 THEN command$ = command$ + letter$ : PRINT
  letter$ ; ELSE PRINT
   UNTIL ascii = 13
```

You can then simplify the string in line 90 since it only needs to contain upper-case words.

SAQ 8

```
20590 DEF PROCaccept_field
20600 REM Puts contents of current field from start to
      cursor position into F$(field)
20610 REM Blank out field to right of cursor
20620 PRINT STRING$(L(field) - chars, " ") ;
20630 F$(field) = LEFT$(F$(field), chars)
20640
20650 REM Strip off blank spaces on right of field entry
20660 stripped = FALSE
20670 REPEAT
20680   IF RIGHT$(F$(field) , 1) = " " THEN F$(field) =
        LEFT$(F$(field), LEN(F$(field)) - 1) ELSE stripped = TRUE
20690   UNTIL stripped
20700 PROCnext_field
20710
20720 ENDPROC
```

Program 6.16

SAQ 9

```
20740 DEF PROCdelete
20750 F$ = F$(field)
20760 len = LEN F$
20770 F$(field) = LEFT$(F$, chars) + RIGHT$(F$, len -
      chars - 1)
20775 REM Blank off old text at right hand side of field
20780 PRINT RIGHT$(F$, len - chars - 1) " " ;
20790 PRINT STRING$(len - chars, CHR$(8)) ; : REM Move
      cursor back
20800 ENDPROC
```

Program 6.17

SAQ 10

```
20960 DEF PROCinsert_close
20970 REM Reconstitute field from left part in F$(field) and
      saved  right part in T$
20980 F$(field) = LEFT$(F$(field), chars + 1)
20990 len = LEN F$(field)
21000 F$(field) = F$(field) + T$ : chars = LEN F$(field)
21010
21020 REM Print new field
21030 PRINT TAB(H2(field),V2(field)) F$(field) ;
21040 REM Blank out field to right of end of field
21050 PRINT STRING$(L(field) - chars, " ") ;
21060
21070 REM Move cursor back to inserting position
21080 PRINT STRING$(L(field) - len, CHR$(8)) ;
21090 chars = len
21110 insert = FALSE
21120 T = 0
21130 max = 1
21140 count = 0
21150 T$ = ""
21160 ENDPROC
```

Program 6.18

SAQ 11

```
  10 REM Address records
  20 REM Fifty records directly accessible for correction
     by PROCinput
  30
  40 MODE 7
  50 maxrecords = 50
  60 DIM H1(5), H2(5), L(5), F$(5),  T$(5), V1(5),  V2(5),
     name$(maxrecords), address1$(maxrecords),
     address2$(maxrecords), address3$(maxrecords),
     tel$(maxrecords)
  70 T$(1) = "Name"
  80 T$(2) = "Address"
  90 T$(3) = T$(2)
 100 T$(4) = T$(2)
 110 T$(5) = "Tel. No."
 120 L(1) = 20
 130 L(2) = 25
 140 L(3) = 25
 150 L(4) = 25
```

```
160 L(5) = 15
170 FOR I = 1 TO 5
180   H1(I) = 1 : H2(I) = 10 : V1(I) = 7 + I : V2(I)
      = V1(I)
190   NEXT I
200
205 PROCstop_garbage
210 REPEAT
220
230   REPEAT
240     CLS : PRINT TAB(0, 10) "Which record do you
        require? " ;
250     INPUT "" R%
260     UNTIL R% >= 1 AND R% <= maxrecords OR R% = 99
270   IF R% = 99 THEN CLS : PRINT TAB(0, 10) "Run ended." : END
280
290   REM Assign record R% to the screen fields
300
310   F$(1) = name$(R%)
320   F$(2) = address1$(R%)
330   F$(3) = address2$(R%)
340   F$(4) = address3$(R%)
350   F$(5) = tel$(R%)
360
370   PROCinput(5, 1 , "ADDRESS FILE")
380
390   REM Assign the screen fields to record R%
400   name$(R%) = F$(1)
410   address1$(R%) = F$(2)
420   address2$(R%) = F$(3)
430   address3$(R%) = F$(4)
440   tel$(R%) = F$(5)
450
460   UNTIL FALSE
470
480 DEF PROCstop_garbage
490 FOR I = 1 TO maxrecords
500     name$(I) = STRING$(L(1), " ")
510     address1$(I) = STRING$(L(2), " ")
520     address2$(I) = STRING$(L(3), " ")
530     address3$(I) = STRING$(L(4), " ")
540     tel$(I) = STRING$(L(5), " ")
550     NEXT I
560 ENDPROC
```

Program 6.19

Notice `PROCstop_garbage` which is executed as soon as the field data has been specified. This procedure fills the empty arrays with strings of spaces equal to the maximum length that will be used in the program. This ensures that there will be enough memory for the strings during the run and that the BBC Micro's memory will not become cluttered with disused, short strings.

UNIT 7

Files

7.1 File commands

7.2 Serial files

7.3 Random access files with one field

7.4 Random access files with multiple fields

7.5 Sorting files

Assignment G

Objectives of Unit 7

Answers to SAQs

7.1 File commands

In this book, we have presumed that you have already learnt a little about files and that you can create and read back a file of, say, ten names. If you are not familiar with such techniques then we suggest that you study either Chapter 31 of the User Guide or Unit 9 of *30 Hour BASIC* before tackling Units 7 and 8 of this course. We shall start this unit with a reminder about the main file commands.

Identifying a file

In a program, a file is specified by two items of information. First, the file must have a name which can be up to 10 letters long on cassette (7 on disk). Second, the file must have a temporary channel number. This is a variable by which the disk file system (DFS) or cassette file system (CFS) identifies a file. The distinction between the name of the file and its channel number is best explained by an example. Suppose that you wish to create a new file `MYFILE`. To do this, you write

```
CH = OPENOUT "MYFILE"
```

The operating system opens a file, which it will name `MYFILE`, but, until the program meets a statement which closes the file, the operating system will identify the file by the number `CH`. (Note that you do not usually need to know what number `CH` is). So

`MYFILE` is *your* name for the file

`CH` is the temporary identification which the operating system uses for your file.

Putting data on the file

To put data on the file, you use the statement

```
PRINT# CH
```

To put `HELLO` onto `MYFILE` you write

```
PRINT# CH, "HELLO"
```

Or, if you want to put a number (say 8327) on the file, you write

```
PRINT# CH, 8327
```

And, when using variables, you write, say,

```
PRINT# CH, A
PRINT# CH, A$
```

Closing a file

Once you have finished putting data onto the file, close the file with the statement

```
CLOSE# CH
```

This involves two steps:

(a) Any remaining characters in the computer's file buffer are transferred to the file. (The file buffer is a small section of memory into which data going to or from the file is put).

(b) The operating system releases the channel, `CH`.

Alternatively, you can use the command

```
CLOSE# 0
```

which closes *all* files currently in use.

Reading from a file

To read from a file, you must first tell the operating system that you wish to read back data from a particular file. You indicate this with the function `OPENIN` as in

```
CH = OPENIN "MYFILE"
```

To read a string, you use

```
INPUT# CH, string$
```

and to read a floating point variable you use

```
INPUT# CH, number
```

A file of ten names

If an array `name$( )` holds ten names then the following procedure would create a file and store those ten names in it.

```
1000 DEF PROCcreate_file
1010 CH = OPENOUT "NAMES"
1020 FOR I = 1 TO 10
1030   PRINT# CH, name$(I)
1040   NEXT I
1050 CLOSE# CH
1060 ENDPROC
```

The contents of the file could be read into an array `data$( )` with the following procedure:

```
2000 DEF PROCread_file
2010 CH = OPENIN "NAMES"
2020 FOR I = 1 TO 10
2030   INPUT# CH, data$(I)
2040   NEXT I
2050 CLOSE# CH
2060 ENDPROC
```

*COMPACT

If you are using disks then there is one other command that you will need – *COMPACT. When you create a file with OPENOUT, there must be enough continuous space on the disk for the file. The BBC disk operating system cannot make up files out of odd bits of free disk space scattered all over the disk. But the system does provide you with *COMPACT, which you use to move all existing files on the disk into one continuous section of disk, leaving all the spare disk space in another continuous section. *COMPACT cannot be used within a program so, before running any program which will create a file, insert the disk in the drive and type

```
*COMPACT <drive number>
```

in immediate mode. After the compact routine is completed, you will be told how many free sectors are left. There must be at least 64 free sectors for you to create a file with OPENOUT. If there fewer sectors left, but enough for your purpose, then you can reserve a file of the required length with

```
*SAVE name 0 length
```

where length is the number of bytes that you require in hex.

*COMPACT cannot be used within a program since it uses the same memory space as programs. Calling *COMPACT from a program will therefore overwrite the program. The best you can do is to make the program execute a disk-full routine (thus destroying itself) and then to automatically restart itself. The following procedure would do this:

```
DEF PROCdisk_full
*SPOOL temp
PRINT "*COMPACT"
PRINT "CHAIN" CHR$34 "progname" CHR$34
*SPOOL
*EXEC temp
*DELETE temp
```

This procedure would be executed by a line in your error handling procedure:

```
IF ERR = 198 THEN PROCdisk_full
```

OPENIN, OPENOUT, PRINT#, INPUT#, CLOSE# and *COMPACT are the minimum file commands that you need in order to create simple cassette or disk files. Let's now see how we can create more sophisticated files by a few extensions to those statements and some ideas on file design.

7.2 Serial files

There are two types of files – serial files and random access files. Both kinds can be created on disk but on tape it is only possible to create serial files although a cumbersome type of random access file on tape can be created using motor control. To access an item of data on a serial file it is necessary to read the whole file from the beginning of the file up to the item required. To access an item of data on a random access file, it is only necessary to know where it is and then you can go direct to that point on the disk. Consequently, serial files are of limited value except when they are short. They are, however, easy to create and manage.

There are few circumstances (if any) where it pays to read a serial file to find one item of data. Instead, a serial file is usually read from start to finish off the tape or disk and into arrays in the computer's memory where the data can then be accessed quickly. Once a serial file has been read into the computer, it can be accessed as if it were a random access file. The limitation on this approach is the amount of memory available in the computer. Whereas a disk or tape might hold 100,000 bytes of data, perhaps only 20,000 or so bytes can be held in the computer at any one time, depending on the amount of memory being used by your program.

Serial files cannot be amended in the way random access files can be. If you load a serial file into the computer and alter one single byte, the only way to correct the file on tape or disk is to write all the data back as a *new* serial file (unless you are using byte-handling file techniques i.e. `BPUT#` and `BGET#`). We have not used these techniques in this book.

The principles of serial and random access file handling are summarised in Figure 7.1.

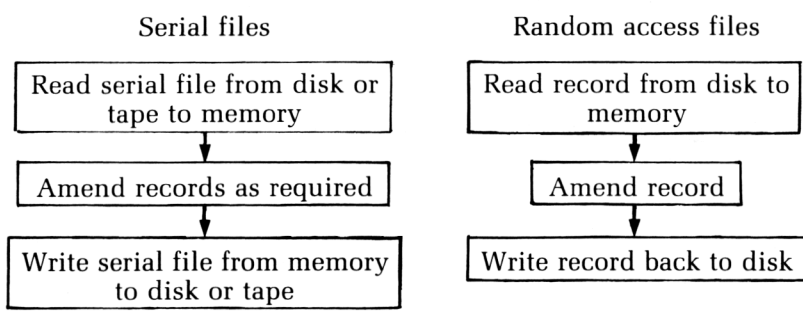

Serial files
Read serial file from disk or tape to memory
Amend records as required
Write serial file from memory to disk or tape
Random access files
Read record from disk to memory
Amend record
Write record back to disk

Figure 7.1 The read/write process for serial and random access files

The concept of a record

Most files consist of records, i.e. collections of data in a pre-determined format. It is possible to create files other than in record format, but we shall first look at record format files only.

A record is the unit of data in a file. If a file consists of a list of names, then each name is a record:

Record 1	Record 2	Record 3
Peter	Mary	Jane

Generally, however, a unit of data consists of several items such as name, address and telephone number:

Record 1	Record 2	Record 3
Peter , York , 234	Mary , Bath , 178	Jane , Norwich , 972

Each of these records consists of three *fields*:

- ☐ name field
- ☐ address field
- ☐ telephone number field.

When we file data in this format, each record is stored on tape or disk with a statement such as

```
PRINT# CH, name$, address$, tel$.
```

More generally, if there are 10 records in the computer held in the arrays `name$( )`, `address$( )` and `tel$( )` then they could be filed in a file `'file$'` by the program segment:

```
100 CH = OPENOUT file$
110 FOR I = 1 TO 10
120   PRINT# CH, name$, address$, tel$
130   NEXT I
140 CLOSE# CH
```

Program 7.1

Program 7.1 will place the data strings onto the tape or disk one after the other:

```
PeterYork234MaryBath178JaneNorwich972
```

(We will explain shortly how the computer finds the beginning of each string).

SAQ 1

Write a program segment that reads back the 10 records of `'file$'` into the arrays `name$( )`, `address$( )` and `tel$( )`.

What we have written in Programs 7.1 and 7.12 will work on disk or tape but there are some differences between BBC disk and tape file statements. All programs in this unit are written in disk format. Unless otherwise stated, programs in section 7.2 run as shown on tape as well. Where they need amending for tape, the tape variations are given. Programs in sections 7.3 to 7.5 will only run on disk.

Some refinements

Programs 7.1 and 7.12 are not safe as they stand. Program 7.1 will crash if an attempt is made to use an invalid file name or where no file of that name is on the disk. (With tape you just wait and wait but nothing happens). It is therefore useful to be able to detect if a file exists and where its end is.

End of file

When reading from a file, it is not necessary to know how many records there are since the file itself has an 'end of file' marker put there by the computer when you wrote the file. The marker is detected by

```
PTR# CH >= EXT#        CH for disk
```

`EXT# CH` has a value equal to the number of bytes in the disk file `CH`. The test `PTR#CH >= EXT# CH` checks whether the file pointer is beyond the end of the file.

and

```
EOF# CH = TRUE         for tape
```

The test `EOF# CH = TRUE` is used to check whether the end of the file marker on the tape has been reached.

A neater way of writing Program 7.12 is:

```
200 CH = OPENIN(file$)
205 I = 1
210 REPEAT
220     IF NOT  PTR# CH  = EXT#  CH THEN  INPUT# CH,  name$(I),
        address$(I), tel$(I)
225   I = I + 1
230   UNTIL PTR# CH = EXT# CH
240 CLOSE# CH
```

```
Tape:
220 IF NOT EOF# CH THEN ... etc
230 UNTIL EOF# CH
```

Program 7.2

Is the file there?

When you use the statement `CH = OPENIN(...)`., the computer assigns a number to `CH` (we don't need to know which number it has chosen) which is the number the computer will use for that file until it is closed. But if there is no file on the disk, the program returns the value zero for `CH`. We can make use of this to stop the program crashing when the user tries to access a file that is not there. A very simple routine is as follows:

```
100 REPEAT
110   CH = OPENIN("file")
120   IF CH = 0 THEN PRINT "No file of that name on the disk.
      Please re-enter file name."
130   UNTIL CH
```

Creating a serial file management program

We are now going to design a serial file management program following the approach to programming set out in Unit 5. We start with analysis.

Analysis

The program should be able to create and display files of records with each record in the form:

Name	(up to 20 characters)
Address	(up to 25 characters)
Address	(up to 25 characters)
Address	(up to 25 characters)
Tel No.	(up to 15 characters)

- ☐ For test purposes, we will limit the file to 10 records.
- ☐ The user must be able to enter names and addresses. (We can use `PROCinput` for this).
- ☐ The user must be able to create a file of addresses under a name of his choice.
- ☐ The user must be able to read back his file and amend the records.
- ☐ The user must be able to save his amended file under the old name or under a new name.

Design

The structure of the program is shown in Figure 7.2.

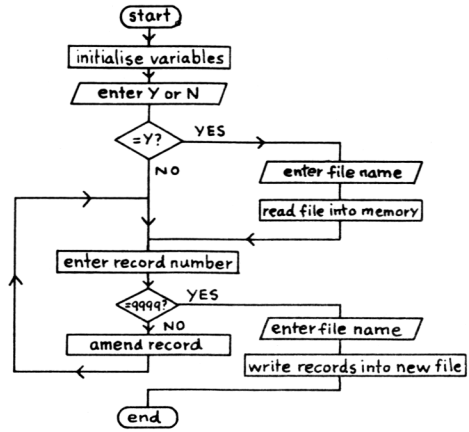


Figure 7.2 The design of the serial file management program

Of these activities, the three major tasks are 'reading the file into memory', 'amending a record' and 'writing the records to a new file'. We already have procedures for displaying and amending a record, so we only need to write procedures for reading the file and writing it back. The next two SAQs give you a chance to try coding these.

SAQ 2

Write a procedure to write to a serial file the records held in the arrays `name$( )`, `address1$( )`, `address2$( )`, `address3$( )` and `tel$( )`.

SAQ 3

Write a procedure to read the file created in SAQ 2 back into the five arrays.

As usual, by building program modules, we can put the final linking material in very simply. Program 7.3 is the controlling section of the serial file management program. Below line 370 we can insert the procedures: `PROCread_file`, `PROCwrite_file`, `PROCinput`.

```
10 REM Address record serial file accessible for
   correction by PROCinput
20
30 MODE 7
40 total = 10 : REM Number of records
50 fields = 5 : REM Number of fields in one record
60
70  DIM H1(fields), H2(fields), L(fields), F$(fields),
    T$(fields), V1(fields), V2(fields), name$(total),
    tel$(total), address1$(total), address2$(total),
    address3$(total)
80 REM Set up PROCinput screen data
```

```
 90 T$(1) = "Name" : T$(2) = "Address" :  T$(3) =  T$(2) :
    T$(4) =  T$(2) : T$(5) = "Tel No." : L(1) = 20 :
    L(2) = 25 : L(3) = 25 : L(4) = 25 : L(5) = 15
100 FOR I = 1 TO fields
110   H1(I) = 10 : V1(I) = 7 + I : V2(I) = V1(I)
120   NEXT I
130
140 REPEAT
150   CLS : PRINT TAB(0, 10) "Do you wish to access an
      existing  file? (Y/N) " ;
160   INPUT "" R$
170   R$ = LEFT$(R$, 1)
180   UNTIL R$ = "Y" OR R$ = "N"
190 IF R$ = "Y" THEN PROCread_file
200
210 REPEAT
220   REPEAT
230     CLS : PRINT TAB(0, 10) "Enter record number
        (9999 to end) " ;
240     INPUT "" record%
250     UNTIL record% >= 1 AND record% <= total OR
        record% = 9999
260   IF record% = 9999 THEN PROCwrite_file :  CLS :
      PRINT TAB(0,  10) "Run ended." : END
270
280   REM Assign record record% to the screen fields
290
300      F$(1) =  name$(record%) :  F$(2) =  address1$(record%)
         :  F$(3) = address2$(record%) : F$(4) =
         address3$(record%) : F$(5) = tel$(record%)
310
320   PROCinput(5, 1, "ADDRESS FILE")
330
340   REM Assign the screen fields to record record%
350        name$(record%)  =  F$(1)  :  address1$(record%)
           =  F$(2)  : address2$(record%)  =  F$(3) :
           address3$(record%) =  F$(4) :  tel$(record%) = F$(5)
360
370   UNTIL FALSE
```

Program 7.3

Non-standard formats

Many books tell you that files must be in record format but this is not the case. You can create a file which contains any sequence of string and numeric variables. You will only be able to read back the file if you know the order in which the various types of variable appear.

Suppose, for example, that you wished to place on a serial file the following data:

Your name	Father Christmas	`name$`
Today's date	251282	`date$`
Countries done	Greenland, Iceland, Norway	`done$( )`
Countries to do	Australia, New Zealand, Mexico	`todo$( )`
Presents given	2075942	`given`
Presents in stock	918433572	`stock`

The following program would write this to a serial file:

```
 10 REM Father Christmas stock record
 20
 30 DIM done$(3),todo$(3)
 40
 50 REM Variables need to be defined in full program
 60
 70 CH = OPENOUT "XMAS"
 80
 90 PRINT# CH, name$
100 PRINT# CH, date$
110
120 FOR I = 1 TO 3
130   PRINT# CH, done$(I)
140   NEXT I
150
160 FOR I = 1 TO 3
170   PRINT# CH, todo$(I)
180   NEXT I
190
200 PRINT# CH, given
210 PRINT# CH, stock
220
230 CLOSE# CH
```

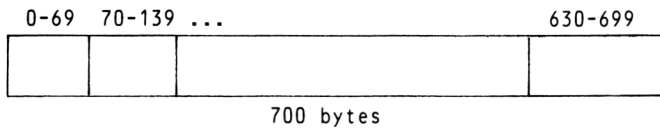
Program 7.4

SAQ 4
Write a program to read back the file `XMAS`.

7.3 Random access files with one field
The programs in this section of the book can only be run if you have a disk system.

The serial file management program is easy enough to use under our test condition of only ten records. But the more records we allow, the longer it takes to read the file at the start of the run and the longer it takes to re-write the file at the end of each run. There is no solution to this problem with serial files and serial files are the only kind that can be sensibly created on cassette.

Random access files are different. They are constructed so that we know where in a file a particular record is, and the disk system provides us with a method of getting to that position in the file. Suppose that we wish to create a random access file of 10 records, each 70 characters long. Each character (a letter, digit, or special character such as space or '.') takes up one byte of file space so we will need 700 bytes of space. (A 'byte' is one unit of file space or one unit of computer memory). The records will be spaced out to each take up 70 bytes:



(In practice, we need a few more bytes to take account of the file structure. For the moment, we will ignore this requirement). Because we know how long each record is, we can say where it is in the file. Record 6 starts at byte 350 i.e. (6 - 1) * 70. Since the disk system allows us to start reading or writing a file at any byte, we can directly read or write record 6. If, say, record 6 was a single string then we would read it with the statements

```
100 PTR# CH = 350
110 INPUT# CH, F$
```

`PTR# CH` (read this as 'pointer file `CH`') tells the disk system to carry out the next disk operation at `PTR# CH` bytes from the start of file `CH`.

To use this to create random access files, it is essential that the records are all of the same length and that we know the length before we try to access any of the records. In this unit, we consider only cases where we know the record length because it is specified in the program. Unit 8 looks at storing the record length in the file itself.

Let's now look more closely at how data is held on the file and how `PTR#` enables us to access it.

Markers on files

Suppose you have a file 1000 bytes long on a disk. With the use of the variable `PTR` you can place a piece of data wherever you want in the file. To put the string `HERE` at the 500th byte, you write:

```
100 PTR# CH = 500
110 PRINT# CH, "HERE"
```

Or, to put the number 1982 at the 600th byte, you write:

```
120 PTR# CH = 600
130 PRINT# CH, 1982
```

When you do this, you don't just write HERE or 1982. For example, PRINT# CH, "HERE" would write bytes on the file as follows:

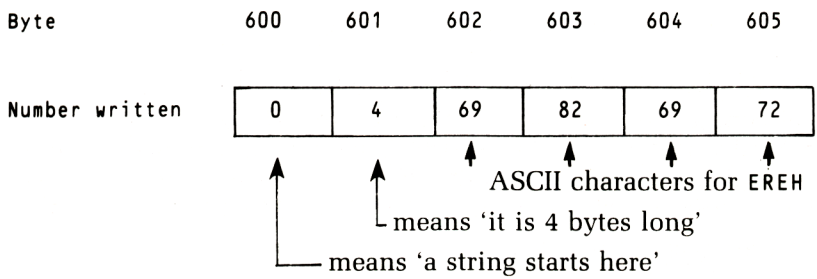


Figure 7.3 The string "HERE" written on a file

A string on the disk needs 2 bytes more space than its length.

K Type and run Program 7.5 to see how this works. The program makes use of the statement 'BGET' which reads in a single byte from a file.

```
 10 REM String filing test
 20
 30 CH = OPENOUT "TEST"
 40 INPUT "Enter a string " L$
 50 PRINT# CH, L$
 60 CLOSE# CH
 70
 80 CH = OPENIN "TEST"
 90 PRINT '' "String read back from file, byte by
    byte:" '
100 PRINT "Byte", "ASCII", "CHR$"
110 I = 0
120 REPEAT
130   X = BGET# CH : PRINT ; I, ; X, CHR$ X
140   I = I + 1
150   UNTIL PTR# CH = EXT# CH
160 CLOSE# CH
```

Program 7.5

```
>RUN
Enter a string HERE
String read back from file, byte by byte:
```

Byte	ASCII	CHR$
0	0	
1	4	
2	69	E
3	82	R
4	69	E
5	72	H

Notes:

(1) The first byte on a disk file is byte 0, not byte 1.

(2) The string itself is placed on the disk back to front. It is automatically restored to its correct order by the computer's filing system so that we can ignore this reversal when we write programs which deal with strings saved and loaded using `PRINT#` and `INPUT#`.

Numbers are slightly different:

Floating point numbers:	take up 6 bytes of the file and start with ASCII 255 e.g. 123.76 would be stored as: 255 184 30 133 119 135.
Integers:	take up 5 bytes of the file and start with ASCII 64 e.g. 123 would be stored as: 64 0 0 0 123

Note: the size of a number does not affect the amount of disk space required to store it.

SAQ 5

Write a program to demonstrate that floating point and integer variables take up respectively 6 and 5 bytes of disk space.

Placing records at known positions

Now that we have established the principle of `PTR` and know how much space variables take up, we can place records at precise locations and retrieve them from those locations.

Example 1

An array `record$( )` holds 10 strings of maximum length 30 bytes. Place the 10 strings in a random access file.

Solution

Each string has to have 30 bytes file space for its contents plus 2 bytes file space for its identifier on the disk. So the strings are placed at

```
 0 bytes
32 bytes
64 bytes etc.
```

The position of string I will be (I - 1) * 32 bytes:

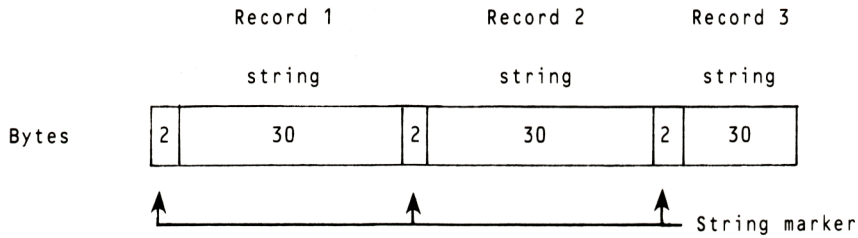


Figure 7.4 Three strings of length 30 on a file

The program to file these records is Program 7.6.

```
140 REM Create file
150 CH = OPENOUT "random"
160 FOR I = 1 TO records
170   PTR# CH = (I - 1) * 32
180   PRINT# CH, record$(I)
190   NEXT I
200 CLOSE# CH
```

Program 7.6

(We are assuming that the program is not using `R$(0)` to hold a record)

When it comes to reading them back, we only read back the records we need. This might be all of them (for a sort or a printout) or it might be just one. To read back one record, we need Program 7.7.

```
230 REM Read from file
240 CH = OPENIN "random"
250
260 REPEAT
270   PRINT TAB(0, 8) "Enter record number (99) to end " ;
280   INPUT "" record
290   UNTIL (record - 1) * 32 < EXT# CH OR record = 99
300 PTR# CH = (record - 1) * 32
310 INPUT# CH, record$
320 CLOSE# CH
330 PRINT record$
```

Program 7.7

Note that we have closed the file (line 320) as soon possible after reading it. This (a) prevents any loss of data in the event of a subsequent file error and (b) safeguards us against accidentally incurring a 'file already open' error message on a subsequent attempt to open the file.

Program 7.7 cannot be used until the file random has been created by Program 7.6. Program 7.6 in its turn needs some data in the array record$( ) to put on the file. This is arranged in Program 7.8 which amalgamates Programs 7.6 and 7.7. First the array record$( ) is filled with test data (lines 110 - 130). Then the test data is put onto a random access file (lines 160 - 210). Finally, the user can access record after record in any order (lines 260 - 350).

```
 10 REM Demonstration of creation of random access file
 20 REM and reading back individual records from it
 30
 40 MODE 7
 50 records = 10 : REM Number of records
 60 DIM record$(records)
 70
 80 REM Fill array with test data to put on file
 90 CLS
100 PRINT TAB(0, 10) "File being created."
110 FOR I = 1 TO records
120   READ record$(I)
130   NEXT I
140
150 REM Create file
160 CH = OPENOUT "random"
170 FOR I = 1 TO records
180   PTR# CH = (I - 1) * 32
190   PRINT# CH, record$(I)
200   NEXT I
210 CLOSE# CH
220 CLS
230
240 REM Read from file
250 CH = OPENIN "random"
260 REPEAT
270
280   REPEAT
290     PRINT TAB(0, 8) "Enter record number (99) to end     "
        ;  TAB(32, 8) ;
300     INPUT "" record
310     UNTIL record >= 1 AND record <= records OR record = 99
320
330   IF record <> 99 THEN PTR# CH = (record - 1) * 32 : INPUT#
      CH, record$ : PRINT record$  STRING$(32 - LEN record$,
      " ")
340   UNTIL record = 99
350 CLOSE# CH
```

```
360
370 CLS : PRINT TAB(0, 10) "Run ended." : END
380
390 REM Test data
400 DATA London, Newcastle, Cambridge, Bath
410 DATA Portsmouth, Dover, Edinburgh, Cardiff
420 DATA Bristol, Norwich
```

Program 7.8

SAQ 6

Can a random access file be read by a serial file read program?

Changing a random access file

Suppose we now want to change a record in the file `random`. To do this we move the file pointer, `PTR`, to the start of the record and then execute the statement

```
PRINT# CH, record$
```

in exactly the same way as when that record was created.

When you change a random access record you must use `OPENIN` and not `OPENOUT`. `OPENOUT` is used only to create the initial file. Once a file exists you must never again address it with `OPENOUT`. Doing so destroys the file.

Program 7.9 is a program to change the records in the file called 'random'.

```
 10 REM Demonstration of method of changing records
 20 REM in random access files
 30
 40 length = 32
 50 file$ = "random"
 60
 70 REPEAT
 80
 90   CH = OPENIN file$
100   REPEAT
110     CLS
120     PRINT TAB(0, 10) "Enter number of record to be changed
        (99 to end) " ;
130     INPUT "" record
140     UNTIL record >= 1 AND (record - 1) * length <
        EXT# CH OR record = 99
150   IF record = 99 THEN PROCexit
160
170   PROCamend
180   CLOSE# CH
```

```
190
200   UNTIL FALSE
210 END
220
230 DEF PROCamend
240 REM Get record from file and display
250   PTR# CH = (record - 1) * length
260   INPUT# CH, record$
270   PRINT ' "Record at present is: " record$
280
290   REPEAT
300     PRINT ' "Enter new record (# to leave unaltered)."
310     INPUT "" record$
320     UNTIL LEN record$ <= length - 2
330
340   REM File new record at position of old
350   PTR# CH = (record - 1) * length
360   IF record$ <> "#" THEN PRINT# CH, record$
370   ENDPROC
380
390 DEF PROCexit
400 CLOSE# 0
410 CLS
420 END
```

Program 7.9

Creating blank records

When you first create a file, you are not likely to be ready to fill each record with data. On the other hand, a file program will crash if there is no record at a location which you attempt to read from with `INPUT# ...` The solution to this problem is to place blank records at every field location on a file when it is first created. If each record is to consist of one string only, then a blank file called `myfile` for 50 records would be created by Program 7.10.

```
 20 REM Create a blank file for
 30 REM 50 records of one string each
 40 REM each record has length of 45 bytes
 50
 50 length = 47
 60 records = 50
 70
 80 CH = OPENOUT "myfile"
 90 FOR I = 1 TO records
100   PTR# CH = (I - 1) * length
110   PRINT# CH, ""
```

```
120   NEXT I
130 CLOSE# CH
140 CLS
150 PRINT TAB(0, 10) "File created."
160 END
```

Program 7.10

The line that is doing the work is line 90, which prints an empty string at the start of each record position. Remember – once a blank file has been created as in Program 7.10, it must never be addressed again with `OPENOUT`; always use `OPENIN` for a disk file that exists.

SAQ 7

Write a program which offers the user a menu:

```
1. Create new file
2. Use a file
```

Typing '1' allows the user to create a blank file to hold 50 records, each consisting of one string of maximum length of 60 bytes. Typing '2' allows the user

(a) to see the contents of any record
(b) leave it as it is
(c) to enter a new string for that record

7.4 Random access files with multiple fields

So far, each record of our random access files has consisted of only one field. Frequently, we will want to have several fields just as we did in Program 7.3 for serial files. There are three ways in which we can place a multiple field record on disk. They are best explained by an example.

Example 2

A record in a file is to consist of three strings:

`first$`	of maximum length 20
`second$`	of maximum length 30
`third$`	of maximum length 25

Place this record on file in random access format as record number 'number'.

Method (a)

This method uses the same principle as that for serial files. The record is printed on as 3 strings, one after the other. The maximum space required will be 20 + 30 + 25 + 2 + 2 + 2 (for 3 strings) = 81 so the record length is 81 bytes:

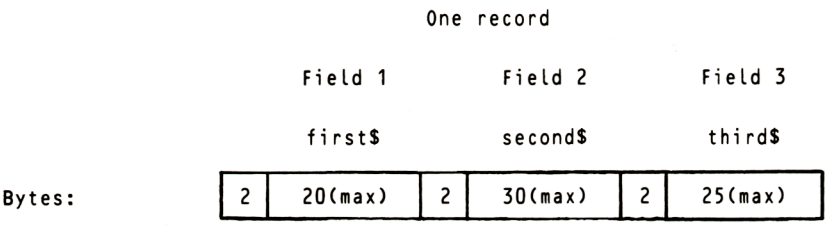


Figure 7.5 A three string record with no space between strings

The statements to write and read the record are:

```
Write    PTR# CH = (number - 1) * 81
         PRINT# CH, first$, second$, third$

Read     PTR# CH = (number - 1) * 81
         INPUT# CH, first$, second$, third$
```

Note: With this method, the whole record must be written in one go and read in one go.

Method (b)

The second method is to turn the three strings into one single string containing the individual strings at positions determined by their maximum length. This is done as follows:

```
record$  =       first$ + STRING$(20 - LEN(first$), " ")
                 + second$ + STRING$(30 - LEN(second$), " ")
                 + third$ + STRING$(25 - LEN(third$), " ")
```

(The statements `STRING$(N, A$)` creates a string consisting of `A$` repeated N times).

So, `record$` is now a string like this:

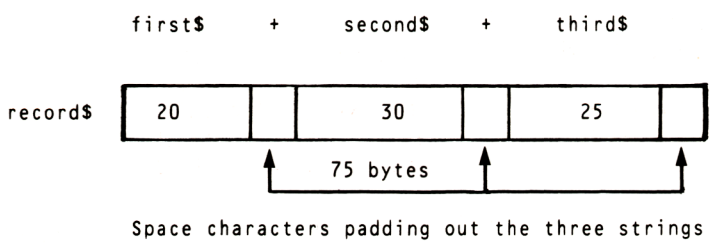


Figure 7.6 Amalgamating three strings into one string

The shaded parts indicate blanks to pad out `first$`, `second$` and `third$` to their maximum lengths of 20, 30 and 25. This string is 75 bytes long and requires 2 further bytes (see Figure 7.2) to put it on disk – a total of 77. The record goes on the file as:

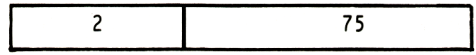


Figure 7.7 A three-string record written as one string

The record can now be field by the statements:

```
Write          PTR# CH = (number - 1) * 77
               PRINT# CH, record$

Read           PTR# CH = (number - 1) * 77
               INPUT# CH, record$
```

This method involves fewer disk operations and is therefore faster than method (a) but it requires tedious packing of the string to go onto the file and unpacking when it comes off. (It also fills the memory with string garbage.) Its value is in saving disk space when a record consists of a lot of short strings. For example, 20 strings of one byte length require 60 bytes of record by method (a) but only 22 bytes by method (b).

Method (c)

This method involves placing each string on the disk at precisely the right point in the record as determined by its maximum length.

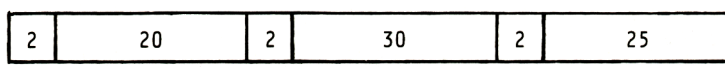


Figure 7.8 Three strings at fixed locations making up one record

The write and read statements are:

```
Write:         PTR# CH = (number - 1) * length
               PRINT# CH, first$
               PTR# CH = (number - 1) * length + field1_len + 2
               PRINT# CH, second$
               PTR# CH = (number - 1) * length + field1_len +
               field2_len + 4
               PRINT# CH, third$

Read:          PTR# CH = (number - 1) * length
               INPUT# CH, first$
               PTR# CH = (number - 1) * length + field1_len + 2
               INPUT# CH, second$
               PTR# CH = (number - 1) * length + field1_len +
               field2_len + 4
               INPUT# CH, third$
```

The constants 81, 22 and 54 are recorded in the first record so that they can be found when we want to use the file.

This is the slowest method but it has one advantage over methods (a) and (b) in that you can read and alter *single* fields. Suppose, for example, you have a massive record of 2000 bytes and, say, 50 fields. Using method (a) or (b) would require re-writing the whole record even to alter one field. Using this method, the one field can be amended without touching the rest of the record.

SAQ 8

Using method (a), write a procedure to write a record of 5 string fields: `name$`, `address1$`, `address2$`, `address3$`, `tel$`, where the maximum lengths of the strings are 20, 25, 25, 25 and 15. Assume that the file name has already been assigned to the variable '`file$`'. (We are developing a re-write of Program 7.3 to make it work for random access files, so you may wish to refer back to it).

SAQ 9

Using method (a), write a procedure to read the record created in SAQ 8.

SAQ 10

Write a program to create a blank file of 30 records for use with Program 7.19.

SAQ 11

Programs 7.18 to 7.20 have begun the modification of Program 7.13 to use it on random access files. Complete the modifications.

7.5 Sorting files

Some files are kept in, say, alphabetical order. For example, you might wish to have an address file for members of a club with the records kept in alphabetical order by name:

```
Record 1          Abbott

Record 2          Burton

Record 3          Clark

...               ...

Record 52         Mills

Record 53         Noble
```

This requires special programming techniques to deal with the problem of new members and deletions. If Mr Arkwright joins the club and Mr

Mills leaves, then the records have to be re-ordered as:

```
Record 1          Abbott

Record 2          Arkwright

Record 3          Burton

Record 4          Clark

...               ...

Record 53         Noble
```

The two techniques (merge and delete) required for maintaining serial files of this type are described in *30 Hour BASIC*. We do not deal with them further in this book.

A more general approach to providing ordered data is to leave the file unordered but to provide a means of listing its contents in an ordered form. This is the problem that we deal with now.

There are two methods of ordering a file: one is to load the whole file into the computer's memory and then sort it as you would sort any other data; the other is to sort the file whilst it is still on disk (this second method cannot be used for tape files). The first method is, of course, limited to files which are small enough to be loaded into the computer's memory.

Sorting in memory

Unless a file has a one-field record, we have a choice of which field to sort by. A file with records of format:

```
Name          Membership No          Address          Post code
```

might be sorted by name, by membership number, or by post code. (There would be no sense in sorting by address). The first field is called the 'key field'. The field we sort by is called the 'sort field'.

SAQ 12

Write a procedure to read the file made by Program 7.21 into memory.

SAQ 13

Write a procedure to sort the file by the first field and to print out the file contents in the sorted order. (Use the sort algorithm from Program 5.1).

SAQ 14

Test the procedures of SAQ 12 and SAQ 13 on the file you made with Program 7.21.

SAQ 15

Modify `PROCsort_file` to allow you to send it a parameter which will be the field by which the file is sorted. Test the new procedure.

Sorting on disk

A memory sort for 30 records is quite fast. It slows down a lot for large numbers of records (e.g. over 100) but that can be dealt with by changing the sort method from the 'bubble' sort to one of the faster algorithms such as the 'quick' sort or the 'heap' sort. These are not described in this book but are described and demonstrated in other BBC software packages, such as the BBC Micro *Toolbox*.

In SAQ 15, you will have found the benefit of needing only one read of the file for any number of sorts. All that speed and convenience is lost when the file is too big to take off disk into memory. The method we are going to describe assumes that the computer's memory is large enough to hold the sort field of the file even though it is not large enough to hold the whole file. The disk sort must not disturb the records. We are not ordering the disk but want to see its data in order on the screen or printout. Therefore we can only read the disk, not write to it. To carry out the sort we set up an array, `sorter%( )` which starts with its entries as `1, 2, 3 ... total` where `total` is the number of records on the disk. We then read the key fields off the disk into the array `key$( )`. `key$( )` is now sorted in the usual way. Every time we interchange elements of `key$( )`, we interchange the corresponding elements of `sorter%( )`. At the end of the sort, `sorter%( )` is in the order in which we need to record the records off the disk if they are to come off in the sorted order of the key field, e.g.

Before sort

Contents of `key$( )`	Contents of `sorter%( )`
Sunday	1
Monday	2
Tuesday	3

After sort

Monday	2 ←	This tells us that in order to read the disk records in alphabetical order, we need to read them in the order record 2, record 3 and then record 1.
Tuesday	3	
Sunday	1	

This method is accomplished by `PROCsort_disk` in Program 7.11.

```
10 REM Sort random access file on one field
20 REM when file is too big to load into memory
```

```
 30
 40 total = 30
 50 fields = 5
 60 length = 120
 70
 80 DIM F$(total, fields), T$(total), sorter%(total),
    key$(total)
 90 file$ = "ADR" : REM Put the name of your new file here
100
110 REPEAT
120
130   REPEAT
140     PRINT "Sort by which field (1-" ; fields ")? " ;
150     INPUT "" field
160     UNTIL field >= 1 AND field <= fields
170
180   PROCload_key_field(field)
190
200   PROCsort_disk
210
220   PROCprint_disk
230
240   UNTIL FALSE
250 END
260
270 DEF PROCload_key_field(key)
280 CH = OPENIN(file$)
290 FOR I = 1 TO total
300   PTR# CH = (I - 1) * length
310   FOR J = 1 TO fields
320     INPUT# CH, T$(J)
330     NEXT J
340   key$(I) = T$(key)
350   NEXT I
360 CLOSE# CH
370 ENDPROC
380
390 DEF PROCsort_disk
400 REM Initialise sorter%( )
410 FOR I% = 1 TO total
420   sorter%(I%) = I%
430   NEXT I%
440
450 REM Sort on chosen field
460 REPEAT
470   exchange = FALSE
```

```
480   FOR I% = 1 TO total - 1
490     IF key$(I% + 1) > key$(I%) THEN PROCswap
500     NEXT
510   UNTIL NOT exchange
520 ENDPROC
530
540 DEF PROCswap
550 LOCAL T%, T$
560 T$ = key$(I%) : key$(I%) = key$(I% + 1) : key$(I% + 1) = T$
570 T% = sorter%(I%) : sorter%(I%) = sorter%(I% + 1) :
    sorter%(I% + 1) = T%
580 exchange = TRUE
590 ENDPROC
600
610 DEF PROCprint_disk
620 REM Read disk in order of sorter%( )
630 CH = OPENIN(file$)
640 FOR I = 1 TO total
650
660   REM Read and print next record
670   PTR# CH = (sorter%(I) - 1) * length
680   FOR J = 1 TO fields
690     INPUT# CH,T$
700     PRINT T$
710     NEXT J
720
730   PRINT
740   NEXT I
750
760 CLOSE# CH
770 ENDPROC
```

Program 7.11

Assignment G

1. Write a serial file management program with the following specification:

(a) Each record is one field only and holds string data (say, a name).

(b) On creation the file holds two dummy names 'AAAAAA' and 'ZZZZZZ'.

(c) Every time a name is entered, it is inserted into its correct alphabetical position in the list of names.

(d) At the end of each data entry session, the list of names is written back to the serial file to create a new serial file.

Objectives of Unit 7

After studying this unit, you should be able to write programs to:

- ☐ Create a serial file where each record consists of several fields.
- ☐ Read back a serial file where the number of fields in each record is known.
- ☐ Amend entries in a serial file.

If you have access to a disk system, you should also be able to:

- ☐ Create random access files with empty fields.
- ☐ Amend a random access file.
- ☐ Sort a random access file which is too large to read into the computer's memory.

Answers to SAQs

SAQ 1

```
200 CH = OPENIN(file$)
210 FOR I = 1 TO 10
220   INPUT# CH, name$(I), address$(I), tel$(I)
230   NEXT I
240 CLOSE# CH
```

Program 7.12

SAQ 2

```
540 DEF PROCwrite_file
550
560 REPEAT
570   letters = TRUE
580   CLS
590   PRINT TAB(0, 10) "Enter name for file " ;
600   INPUT "" file$
610   FOR I = 1 TO LEN file$
615     t = ASC MID$(file$, I, 1)
620     IF t < ASC "0"  OR t > ASC "z" OR (t > ASC "9" AND t <
        ASC "A") THEN letters = FALSE
630     NEXT I
640   UNTIL LEN file$ <= 7 AND letters
650
660 REM Write all records onto file
670 CH = OPENOUT file$
680 FOR I = 1 TO total
690   PRINT# CH, name$(I), address1$(I), address2$(I),
      address3$(I), tel$(I)
```

```
700   NEXT I
710
720 CLOSE# CH
730 ENDPROC
```

Program 7.13

SAQ 3

```
370 DEF PROCread_file
380 REM Find and open file
390 REPEAT
400
410   CLS
420   PRINT TAB(0, 10) "Enter name of file " ;
430   INPUT "" file$
440   CH = OPENIN file$
450   UNTIL CH
460
470 REM Read all records off file into memory
480 I = 1
490 REPEAT
500    IF NOT  PTR# CH  = EXT#  CH THEN  INPUT# CH,  name$(I),
       address1$(I), address2$(I), address3$(I), tel$(I)
510   I = I + 1
520   UNTIL PTR# CH = EXT# CH
530
540 CLOSE# CH
550 ENDPROC
```

Program 7.14

Tape:
Delete lines 390 and 450.
Line 480 `IF NOT EOF# CH THEN ...`
Line 500 `UNTIL EOF# CH`

SAQ 4

```
 10 REM Program to read back Father Christmas data
 20
 30 DIM done$(3),todo$(3)
 40
 50 CH = OPENIN "XMAS"
 60
 70 INPUT# CH, name$
 80 INPUT# CH, date$
 90
100 FOR I = 1 TO 3
110   INPUT# CH, done$(I)
120   NEXT I
```

```
130
140 FOR I = 1 TO 3
150   INPUT# CH, todo$(I)
160   NEXT I
170
180 INPUT# CH, given
190 INPUT# CH, stock
200
210 CLOSE# CH
```

Program 7.15

SAQ 5

```
 10 REM Floating point numbers on files
 20
 30 INPUT "Enter a number " N
 40 CH = OPENOUT "TEST"
 50 PRINT# CH, N
 60 CLOSE# CH
 70
 90 PRINT '' "Floating point number read back from
    file, byte by byte:" '
100 PRINT "Byte", "ASCII"
105 CH = OPENIN "TEST"
110 I = 0
120 REPEAT
130   X = BGET# CH
140   PRINT ; I, ; X
150   I = I + 1
160   UNTIL PTR# CH = EXT# CH
170 CLOSE# CH
```

Program 7.16

Test run

```
>RUN
Enter a number 123.76

Floating point number read back from file, byte by byte:
Byte        ASCII
0           255
1           184
2           30
3           133
4           119
5           135
```

If you change N to N% in lines 40 and 50, you can examine the method of storing integers on disk.

SAQ 6

No. The serial file read program always expects the next item of data on a file to follow immediately after the last item. With a random access file there may be a jump between the end of the data in one record and the start of the next record.

SAQ 7

```
 10 REM Creation of file of 50 random access records
 20 MODE 7
 30
 40 records = 50
 50 length = 62
 60
 70 REPEAT
 80
 90   REM Menu
100   CLS
110   PRINT TAB(18, 4) "MENU"
120   PRINT TAB(0, 9) "1 Create a new file"
130   PRINT TAB(0, 10) "2 Access a file"
140   PRINT TAB(0,11) "3 End"
150   PRINT TAB(0, 20) "Enter number "
160
165   REM Choose menu option
170   REPEAT
175     PRINT TAB(14, 20) "          " TAB(14, 20) ;
180     INPUT "" option
190     IF option = 1 THEN PROCcreate_file
200     IF option = 2 THEN PROCaccess_file
210     IF option = 3 THEN PROCexit
215     UNTIL option >= 1 AND option <= 3
220
230   UNTIL FALSE
240 END
250
260 DEF PROCcreate_file
270 REM Create file of 50 blank records
280 CLS
290 INPUT TAB(0, 10) "Enter name for new file " file$
310
320 CH = OPENOUT file$
330 FOR I% = 1 TO records
340   PTR# CH = (I% - 1) * length
350   PRINT# CH, ""
```

```
360   NEXT I%
370 CLOSE# CH
380
390 CLS
400 PRINT TAB(0, 10) "File created. Press space bar to
    continue."
410 REPEAT : UNTIL GET$ = " "
420
430 ENDPROC
440
450 DEF PROCaccess_file
460 REPEAT
470   CLS : INPUT TAB(0, 10) "Enter name of file " file$
480   CH = OPENIN(file$)
490   UNTIL CH
500
510 REPEAT
520
530   REM Enter number of record to be accessed
540   REPEAT
550     CLS
560     PRINT TAB(0, 10) "Enter record number (99 to end): " ;
570     INPUT "" record
580     UNTIL record >= 1 AND record <= records OR record = 99
590
600   IF record = 99 THEN CLOSE# CH : PROCexit
610
620   REM Get record and display for amendment
630   PTR# CH = (record - 1) * length
640   INPUT# CH, record$
650   PRINT "Current entry is:" : PRINT ' record$
660   INPUT "Enter new record (press Return to leave unaltered) "
      record$
670   IF record$ <> "" THEN PTR# CH =  (record -  1) *  length :
      PRINT# CH, record$
680
690   UNTIL record = 99
700 CLOSE# CH
710 ENDPROC
720
730 DEF PROCexit
740 CLOSE# 0
750 CLS
760 PRINT TAB(0, 10) "Program run ended."
770 END
780 ENDPROC
```

Program 7.17

SAQ 8

The record length is 20 + 25 + 25 + 25 + 15 + 2 + 2 + 2 + 2 + 2 = 120 bytes.

```
430 DEF PROCwrite_file
440 CH = OPENIN(file$)
450 PTR# CH = (record% - 1) * length
460 PRINT# CH, name$, address1$, address2$, address3$,
    tel$
470 CLOSE# CH
480 ENDPROC
```

Program 7.18

SAQ 9

```
300 DEF PROCread_file
310 CH = OPENIN(file$)
320 PTR# CH = (record% - 1) * length
330 INPUT# CH, name$, address1$, address2$, address3$,
    tel$
340 CLOSE# CH
350 ENDPROC
```

Program 7.19

SAQ 10

```
 10 REM Blank file creation
 20 INPUT "Name for new address file " file$
 30 CH = OPENOUT(file$)
 40 FOR I = 1 TO 30
 50   PTR# CH = (I - 1) * 120
 60   PRINT# CH, "", "", "", "", ""
 70   NEXT I
 80 CLOSE# CH
 90 PRINT "File created."
```

Program 7.20

SAQ 11

```
 10 REM Address record random file
 20
 30 MODE 7
 40 total = 30 : REM Number of records
 50 fields = 5 : REM Number of fields
 60 length = 120 : REM record length
 70
 80 REM PROCinput screen data
```

```
 90  DIM  H1(fields),  H2(fields),  L(fields),   F$(fields),
 T$(fields), V1(fields), V2(fields)
100 T$(1) = "Name" : T$(2) = "Address" :  T$(3) =  T$(4) :
    T$(4) =  T$(2) : T$(5) = "Tel No."
110 L(1) = 20 : L(2) = 25 : L(3) = 25 : L(4) = 25 : L(5) = 15
120 FOR I = 1 TO fields
130   H1(I) = 1 : H2(I) = 10 : V1(I) = 7 + I : V2(I) = V1(I)
140   NEXT I
150
160 REM Enter name of file to be accessed
170 REPEAT
180   CLS
190   INPUT TAB(0, 10) "Enter name of file to be accessed. "
      file$
200   CH = OPENIN(file$)
210   UNTIL CH <> 0
215 CLOSE# CH
220
230 REPEAT
240
250   REPEAT
260     CLS
270     INPUT TAB(0, 10) "Enter record number (99 to end) "
        record
280     UNTIL record >= 1 AND record <= total OR record = 99
290
300   IF record = 99 THEN CLS : PRINT TAB(0, 10) "Run ended."
      : END
310
320   PROCread_file
330
340   PROCinput(fields, 1, "ADDRESS FILE")
350
360   PROCwrite_file
370
380   UNTIL FALSE
390
400 DEF PROCread_file
410 CH = OPENIN(file$)
420 PTR# CH = (record - 1) * length
430 FOR I = 1 TO fields
440   INPUT# CH, F$(I)
450   NEXT I
460 CLOSE# CH
470 ENDPROC
480
```

```
  490 DEF PROCwrite_file
  500 CH = OPENIN(file$)
  510 PTR# CH = (record - 1) * length
  520 FOR I = 1 TO fields
  530   PRINT# CH, F$(I)
  540   NEXT I
  550 CLOSE# CH
  560 ENDPROC

20000 PROCinput( ... ) etc
```

Program 7.21

SAQ 12

```
  500 DEF PROCload_file
  510 CH = OPENIN(file$)
  520 FOR I = 1 TO total
  530   PTR# CH = (I - 1) * length
  540   FOR J = 1 TO fields
  550     INPUT# CH, F$(I, J)
  560     NEXT J
  570   NEXT I
  580 CLOSE# CH
  590 ENDPROC
```

Program 7.22

Notice that in order to make the sort easier, we have abandoned the clear long variable names and put all the data into a two dimensional array, `F$(30,5)`. This needs to be dimensioned at the start of the program.

SAQ 13

```
  610 DEF PROCsort_file
  620 LOCAL I%, exchange
  630 REPEAT
  640   exchange = FALSE
  650   FOR I% = 1 TO total - 1
  660     IF F$(I% + 1, 1) < F$(I%, 1) THEN PROCswap
  670     NEXT
  680   UNTIL NOT exchange
  690 ENDPROC
  700
  710 DEF PROCswap
  720 LOCAL T$, J%
  730 FOR J% = 1 TO fields
```

```
740     T$ = F$(I%, J%) : F$(I%, J%) = F$(I% + 1, J%) :
        F$(I% + 1, J%) = T$
750     NEXT
760 exchange = TRUE
770 ENDPROC
780
790 DEF PROCprint_list
800 LOCAL I, J
810 REM Print out sorted records
820 FOR I = 1 TO total
830   FOR J = 1 TO fields
840     PRINT F$(I, J) " " ;
850     NEXT J
860   PRINT
870   NEXT I
880 ENDPROC
```

Program 7.23

SAQ 14

To test the sort, you need the extra lines:

```
 10 REM TEST ROUTINE FOR SAQ 12 AND SAQ 13
 20
 30 file$ = "ADR"
 40 length = 120
 50 total = 30
 60 fields = 5
 70 DIM F$(total, fields)
 80
 90 PROCload_file
100
110 PROCsort_file
120
130 PROCprint_list
140
150 END
```

SAQ 15

Change `PROCsort_field` to:

```
310 DEF PROCsort_file(field)
320 LOCAL I%, exchange
330 REPEAT
340   exchange = FALSE
350   FOR I% = 1 TO total - 1
360     IF F$(I% + 1, field) < F$(I%, field) THEN
        PROCswap
```

```
370     NEXT
380   UNTIL NOT exchange
390 ENDPROC
```

To test the procedure, modify lines 10 to 120 to read:

```
 10 REM TEST ROUTINE FOR SORTING ON CHOSEN FIELD
 20
 30 file$ = "ADR" : REM Insert the name of your file
    in place of ADR
 40 total = 30
 50 fields = 5
 60 length = 120
 70 DIM F$(total + 1, fields)
 80
 90 PROCload_file
100
110 CLS
120 REPEAT
130   PRINT TAB(0, 10) "Enter number of sort field
     (1-"; fields ") " ;
140   INPUT "" field
150   UNTIL field >= 1 AND field <= fields
160 PROCsort_file(field)
170
180 PROCprint_list
190
200 END
```

Program 7.25

UNIT 8

Data bases

8.1 Towards data-base management

8.2 Designing the program

8.3 Writing the program

Summary: the full program

Assignment H

Objectives of Unit 8

Answers to SAQs

8.1 Towards data-base management

Data-base management is an imprecise term coined to describe certain types of software designed for use in business. These programs enable non-programmers to create files to their specification, enter and amend the data in the files and then to search and sort the files against specifications of the user's choosing. Such a system will generally require man-years of programming to develop and the program itself will retail for £100 to £200 and upwards. Whilst we can't attempt to develop a program of such complexity, we can develop a program which not only illustrates some of the principles of data-base management but which is a thoroughly reliable, simple to use, data-base system. Additionally, because of its structure, it is easy to add new features. The suggestions at the end of the unit, together with the assignment, move in that direction.

Program specification

The system we are going to develop has the following specification:

- ☐ The user may create files of up to 500 records.
- ☐ Each record can have 1 to 20 fields.
- ☐ Fields can be up to 255 bytes long.
- ☐ Fields will all be in string format.
- ☐ The user may access any file created by the system.
- ☐ The user can amend a record.
- ☐ The user can search the file, specifying:
    - ☐ The fields he wishes to consider
    - ☐ The order in which the fields are to appear in the output
    - ☐ Whether the output is to be sorted on the chosen key field
    - ☐ Whether the sort is to be in ascending or descending order.

Principles

In designing a system, there are two points of which we have to take account. First, the file we shall want to create will usually be too big to load into the memory of the computer. The data must therefore be manipulated whilst on disk as far as is possible. Second, the user can create an unlimited number of different files. So the system must have some way of recognising the structure of a file which it is asked to open. We shall now look at how this is done.

There are two basic methods for enabling a program to recognise the structure of a file. Both are implemented at the time the file is created. One method is to create a separate small file alongside the main one which describes the structure of the main file. Each main file is

accompanied by a small 'descriptor' file. The other method (the one that we shall use) is to reserve a part of the main file to describe itself. This portion is called a 'file header' and is created as follows.

In our system, we reserve the first 500 bytes (0-499) of the file for the header. The first real record will always start at byte 500 instead of at the usual byte 0. The header information is derived from the details that the user specifies when he creates the program. It will be:

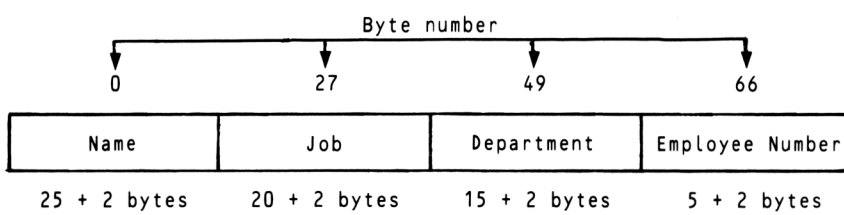
Information	*Variable name*
Name of file	`file$`
Number of records	`number`
Length of each record	`length`
Number of fields in a record	`fields`
Then, for each field:	
Name of field	`N$(I)`
Length of field (field length)	`FL(I)`
Position in record (field pointer)	`FP(I)`

(There is no standard header for files, so this system which we are describing will only work with the programs which we create for it).

This is best illustrated by an example.

Example 1
Work out the record structure and file header details for a staff records file with 50 records of 4 fields: name (25 bytes), job (20 bytes), department (15 bytes) and employee number (5 bytes).

Solution
Each record will be of the form:



The record description for the header in our system is:

```
N$(1) = "Name"
N$(2) = "Job"
N$(3) = "Department"
N$(4) = "Employee Number"
FL(1) = 25
FL(2) = 20
```

```
FL(4) = 5
FP(1) = 0
FP(2) = 27              (= FP(1) + FL(1) + 2)
FP(3) = 49              (= FP(2) + FL(2) + 2)
FP(4) = 66              (= FP(3) + FL(3) + 2)
```

SAQ 1

What is the information that would be needed to make up the file header for Example 1 above. Make sure that you list the information in the correct order. Assume that the file name is 'STAFF'.

8.2 Designing the program

Program design

Following the principles of top-down design, we shall start the design by specifying the main options available to the user. (They will eventually form the basis of the program's menu). There are four main options:

Design stage 1

Create new file

User to be able to input the details of the file that he wishes to create (number of fields, field titles, field lengths, number of records, the name for the file).

Enter data

User to be able to call up any file previously created and enter a record number. The record will be displayed with field titles and the user will be able to amend the record. After amendment, the revised record should be written back to the disk.

Select data

The user should be able to call up any file previously created and see the fields available. He should then be able to specify which fields he wants in his output and the order in which the fields are to appear in each record printed out. He should then be able to specify whether he wants to sort the records on the first field he has chosen or not. He should be given a choice of sorting in ascending or descending order.

End run

We don't need to consider the last option in any more detail at this stage, but the design needs to go into more detail on the other three options. Let's take each in turn and break it down into its constituent routines.

Design stage 2

Create new file

Enter number of fields required
Enter field titles
Enter field lengths
Enter number of records
Enter file name
Save file header to disk
Fill file with blank records

Enter data

Enter name of file
Read file header
Enter record number
Read record
Display record
Amend record (optional)
Write new record to file

Select data

Enter name of file
Read file header
Choose fields for selection
Specify sorting option required
Sort records
Read and print out records in sorted order

Design stage 3

There is enough detail there to be able to begin to create procedures. We could therefore start writing a procedure for each of the steps above. However, we can improve the design by establishing the program structure in terms of the procedures and the order in which the procedures will be called.

Procedure	Action
`PROCdefine_file`	Enter number of fields required Enter field titles and lengths
`PROCcreate_file`	Enter number of records Enter name for file Create file header Create blank records
`PROCamend_record`	Enter record number Read record from disk (use `PROCread_write_record`)

	Display record (use PROCinput) Amend record (optional) Write record to disk (use PROCread_write_record)
PROCread_header	Enter name of file to be accessed Read header details
PROCread_write_record	Read or write a record (Read/write selected by parameter sent to procedure)
PROCselect_data	Choose fields (use PROCselect_fields) Choose sort option (use PROCsort) Print selection (use PROCprint_out)
PROCsort	Choose sort/no sort If 'sort' is chosen then choose ascending or descending order Load key field of all records Sort on key field
PROCprint_out	Print selected fields in sorted order

Design stage 4

Once we have decided on the procedures that we will use, we can refer to stage 1 and expand the design in terms of procedures. So stage 1 finally looks as follows:

Create new file

```
PROCdefine_file
PROCcreate_file
```

Enter data

```
PROCheader
PROCamend_record   (uses PROCread_write_record)
                   (uses PROCinput)
```

Select data

```
PROCread_header
PROCselect_data    (uses PROCselect_fields)
                   (uses PROCsort)
                   (uses PROCprint_out)
```

Design stage 5

At this point, the design has been refined far enough for us to move to coding. But it is useful to distinguish between two types of coding:

Coding the control structure
Coding the procedures

The design itself gives us a very clear prescription for writing the control structure of the program. It will be:

```
160   REM Menu
170   CLOSE# 0
180   CLS
190   PRINT TAB(8, 2) "FILE MANAGEMENT PROGRAM"
200   PRINT TAB(5, 5) "1 Create a new file"
210   PRINT TAB(5) "2 Use a file"
200   PRINT TAB(5) "3 Select data"
230   PRINT TAB(5) "9 End run"
240
250   REM Select option
260   PRINT TAB(0,22) ; "Enter number " ;
270   INPUT "" X
280   IF X = 1 THEN PROCdefine_file : PROCcreate_file
290   IF X = 2 THEN PROCread_header : PROCamend_record
300   IF X = 3 THEN PROCread_header : PROCselect_data
310   IF X = 9 THEN PROCexit
320
330   UNTIL FALSE
```

Program 8.1

8.3 Writing the program

We can now write the program, procedure by procedure. The skeleton of the program is:

```
 350 DEF PROCdefine_file
 850 ENDPROC

 870 DEF PROCcreate_file
1160 ENDPROC

1180 DEF PROCamend_record
1400 ENDPROC

1420 DEF PROCselect_data
1500 ENDPROC

1520 DEF PROCread_header
1660 ENDPROC

1680 DEF PROCread_write_header(record, Z)
1770 ENDPROC
```

```
1790 DEF PROCselect_fields
2000 ENDPROC

2020 DEF PROCsort
2300 ENDPROC

2370 DEF PROCprint_out
2570 ENDPROC

2590 DEF PROCprinter_on
2640 ENDPROC
```

We discuss the new ideas in the main part of the Unit, leaving the others for you as SAQs.

Creating the file

SAQ 2

The variables which specify a particular file are:

`file$`	Name of file
`number`	Number of records
`length`	Length of each record
`fields`	Number of fields in a record
`N$( )`	Names of the fields
`FL( )`	Lengths of the fields
`FP( )`	Position of each field in the record

Write `PROCcreate_file` which should:

- ☐ Ask the user for the number of records he wants on the file.
- ☐ Ask the user for the name of the file.
- ☐ Save the file header onto the disk.
- ☐ Create the blank records on the disk.

(All the other variables will have been defined before entry to this procedure).

K As this unit progresses you can build up the program. Start by loading `PROCinput( ... )` at lines 20000 onwards. Then add the following `DIM` line (we will explain each variable as it is introduced).

```
110 DIM H1(N), H2(N), L(N), F$(N), V1(N), V2(N), T$(N),
N$(maxfields), FL(maxfields), FP(maxfields), pikpnt(maxfields),
piklen(maxfields), pikname$(maxfields), sort$(maxrecords),
sorter(maxrecords)
```

K Now type in `PROCcreate_file` as in the answer to SAQ 2.

Defining the file

Logically this comes before creating the file but you need to know what data is required for creation in order to supply it in the define stage. The define routine, `PROCdefine_file`, allows the user to specify:

- ☐ How many fields he wants (maximum 20).
- ☐ The name for each field (maximum 8 characters).
- ☐ The length of each field (maximum 255 characters).

Nothing new is involved here – only some careful use of indices, since we shall have up to 40 fields on the screen which will look like this when the user is defining his file:

```
        Field lengths and files

Field 1  :        :    Length :   :
Field 2  :        :    Length :   :
Field 3  :        :    Length :   :
etc.
```

Describing the next bit is tricky because there are fields on the screen defining characteristics of fields to be on the file. If the screen fields are numbered:

```
1                2
3                4
5                6
etc.
```

then the odd-numbered fields on the screen hold the field names for the file and the even-numbered fields on the screen hold the lengths of those fields. Here then is the procedure to set up the screen for the user to define his file, allowing him to enter his information and then to transfer it to the variables used in connection with the file header:

```
350 DEF PROCdefine_file
360 LOCAL I, T, ok
370 REM Set up screen data for input
380
390 REPEAT
400   CLS
410   PRINT TAB(0,10) "Enter number of fields
      required on file. " ;
420   INPUT "" fields
430   UNTIL fields >= 1 AND fields <= maxfields
440
450 REM  Set up screen data for field specification
460 FOR I = 1 TO (maxfields * 2) -1 STEP 2
```

```
470   H1(I) = 1 : H2(I) = 13 : V1(I) = 3 + (I + 1) DIV 2 :
      V2(I) = V1(I)
480   H1(I + 1) = 25 : H2(I + 1) = 35 :
      V1(I + 1) = V1(I) : V2(I + 1) = V1(I + 1)
490   T$(I) = "Field " + STR$((I + 1) DIV 2) : T$(I + 1)
      = "Length"
500   L(I) = 8 : L(I + 1) = 3
510   NEXT I
520
530 REM Enter field names and lengths
540
550 REPEAT
560   PROCinput(2 * fields, 1, "Field titles and lengths")
570   ok = TRUE
575 REM Check and trap invalid field lengths
580   FOR I = 2 TO 40 STEP 2
590     IF VAL(F$(I)) > 255 THEN ok = FALSE : F$(I) = "***"
600     NEXT I
610   UNTIL ok
620
630 REM Transfer screen fields to N$( ) and FL( )
640
650 REM  Transfer names to N$( )
660 FOR I = 1 TO 2 * fields - 1 STEP 2
670   N$((I + 1)  DIV 2) = F$(I) : FL((I + 1)  DIV 2) =
      VAL(F$(I + 1))
680   NEXT I
690
700 REM Calculate field pointer positions
710 FOR I = 1 TO fields
720   FP(I) = 0
730   NEXT I
740 FOR I = 2 TO fields
750   FP(I) = FP(I - 1) + FL(I - 1) + 2
760   NEXT I
770
780 REM Calculate record length
790 T = 0
800 FOR I = 1 TO fields
810   T = T + FL(I)
820   NEXT I
830 length = T + 2 * fields
840
850 ENDPROC
```

Program 8.2

K Run the program so far and try creating a file named TEST with 10 records and 3 fields. Make the field names Name (length 20), Address (length 25), P code (length 10). At this stage, you cannot be certain that the file has been correctly created because you don't have a read routine. However, there are three ways of seeing whether things are looking right:

- ☐ First, if there was no error message during the create routine then things look hopeful.
- ☐ Second, after the run, try the following test in immediate mode (i.e. without a line number):

```
>CH = OPENIN "TEST" : PRINT EXT# CH : CLOSE# CH
```

- ☐ You should get the message 1100. If you get an answer that is incorrect for the expected length of file which you have created then something went wrong during the creation process.
- ☐ Third, use *DUMP. This will list all the bytes of the file in hexadecimal on the screen. For an empty file you will mostly get zeros but at least that confirms that the file is there.

Accessing a file

We have now reduced the process of creating a file to a very simple, unskilled, operation that can be carried out by computer users rather than computer programmers. From now on, we want the file to be accessible without the user having to do anything other than specify its name. But the program cannot read the records without details of the structure of the file. For this reason, every time the user wants to access a file, the program must read the file header.

SAQ 3

Write a procedure 'PROCread_header' to ask the user for the name of the file he wishes to access and to read that file's header.

Changing records

To change a record, the program must first read the file header and then move on to the detailed system for amending records. We shall allow the user to alter as many records as he requires before returning to the menu. In the final version of the program, we shall add the line:

```
130 ON ERROR GOTO 140
```

so that wherever the user is in the program, he can return to the menu by pressing escape. (Hence the need for CLOSE# 0 in line 170). Do not add this error trap now since it will prevent you from effectively de-bugging the program. Add it when you are convinced the program is error-free.

We can now consider the detail of `PROCamend_record` which requires three stages:

- ☐ read record
- ☐ display and amend
- ☐ write record.

You can see this cycle in Program 7.17 of Unit 7 but we are going to simplify the coding and write a single procedure that will both read and write the record. It does one or the other according to the value of a flag, Z. If Z = 0, the procedure reads the record. If Z = -1, the procedure writes the record. The procedure is called `PROCread_write_record(record, Z)` where `record` is the record number concerned. Having this one procedure results in a very simple definition of the procedure to amend a record.

```
1180 DEF PROCamend_record
1190 REPEAT
1200
1210   REPEAT
1220     CLS : PRINT TAB(0, 10) "Enter record number " ;
1240     INPUT "" record
1250     UNTIL record >= 1 AND  record <= number
1260
1270   PROCread_write_record(record, 0)
1280
1290   REM Set screen data for PROCinput
1300   FOR I = 1 TO fields
1310     H1(I) = 1 : H2(I) = 11 : V1(I) =  3 +  I :  V2(I) =  V1(I)
         :  L(I) = FL(I) : T$(I) = N$(I)
1320     NEXT I
1330
1340   PROCinput(fields, 1, "RECORD ENTRY " + STR$(record))
1350
1360   PROCread_write_record(record, -1)
1370
1380   UNTIL FALSE
1390
1400 ENDPROC
```

Program 8.3

K Type `PROCamend_record`. You can't test it until we have written `PROCread_write_record(record, Z)`.

SAQ 4

Write `PROCread_write_record(record, Z)`.

[K] Add PROCread_write_record to your program. You should now find that you can fill the records on the file TEST and amend them as you wish. Make up the 10 records of TEST with your own data or with ours below. We shall use our data to check out the selection and sort processes later in the unit.

Sample data for test file, TEST
(These names and addresses are fictitious).

```
Jones
89 Redwood Road, Leeds
L34 5TR

Smith
23 Highbury Avenue, Bath
B54 K98

Andrews
Mill House, Portsmouth
P12 K44

King
156 Long Lane, Blackpool
B6 P31

Morris
60 Jubilee Road, Exeter
EX7 J66

Lansbury
Manor House, Cardiff
C23 V55

Williams
2 Willis Road, Edinburgh
ED4 S62

Oliver
6 Cornwall Road, Bingley
C8 F5

Dearden
9 Rally Road, Birmingham
B8 9TG

Cecil
Rose Cottage, Norwich
NR6 12HR
```

The program we have written so far will create files and allow the user to enter data, amend data and insert individual records. In day-to-day information processing, that part is the preparatory work. The serious business starts with processing and analysing the data. We now look at the development of that part of the program.

When the user wants to select data, he chooses Option 3 from the menu. He is asked to enter the name of his file. The program reads the file header so that it has the file structure data needed for `PROCselect_data`.

`PROCselect_data` can be expanded as much as you like (subject only to memory limitations) since we will simply define it as a series of procedures. We shall start with these:

```
DEF PROCselect_data
    PROCselect_fields
    PROCsort
    PROCprint_out
ENDPROC
```

This is fairly self-explanatory. First, we ask the user to specify the fields from which he wants to extract data:

```
SELECT FIELDS (Enter 1, 2, 3 to select)
NAME      :2
ADDRESS   :1
TEL NO    :3
```

He can choose these in any order, specifying which is to become the key field for the selection. Second, we ask him whether he wants to sort on the key field:

```
Do you want the record sorted on the key field?
```

Enter:

```
N for no
A to sort in ascending order
R to sort in reverse order
```

If he says he does want a sort, the sort is done. Third, his selection is printed out. These procedures are the minimum needed to make the full program of much use. Once constructed, it is easy to add your own procedures to `PROCselect_data` to make the program do things you need.

Choosing fields

This is the most complicated part of the data selection process. Consider a file with records of the form:

```
N$(  )     Name      Dept      Code
FL( )      20        10          5
FP( )       0        22         34
```

If the user wants to print out the file under the headings `Dept` and `Name` then the new key field becomes `Dept` and the new field 2 becomes `Name`. The total number of fields becomes 2 instead of 3. Additionally, the field pointers and field lengths have to be swapped around to fit the new order of the fields. In other words, we must prepare ourselves for the user choosing, from all the fields available, a new set of fields which we will call:

`pikname$( )`	(names of selected fields)
`piklen( )`	(field lengths of selected fields)
`pikpnt( )`	(field pointers of selected fields)

And

`pikf`	Number of fields selected

SAQ 6

What is the purpose of `pikf`, `pikname$( )`, `piklen( )` and `pikpnt( )` in the above 'Name, Dept., Code' case?

The following procedure allows the user to choose his fields, specifying the order for the selected fields. The procedure then sets all the selection variables to fit the user's choice.

```
1790 DEF PROCselect_fields
1800
1810 REM Set up data for screen
1820 FOR I = 1 TO fields
1830   H1(I) = 0 : H2(I) = 12 : V1(I) = 2 + I : V2(I) =
       V1(I) : T$(I) = N$(I) : L(I) = 1 : F$(I) = ""
1840   NEXT I
1850
1860 PROCinput(fields, 1, "SELECT FIELDS (Enter 1,2,3
     to select)")
1870
1880 REM Calculate the number of fields chosen (pikf)
1890 pikf = 0
1900 FOR I = 1 TO fields
1910   IF VAL F$(I) >= 1 AND  VAL F$(I) <= fields THEN
       pikf = pikf + 1
1920   NEXT
1930
1940 REM Find names of chosen fields (pikname$( )),
     lengths (piklen(  )) and pointer positions (pikpnt( ))
1950 FOR I = 1 TO pikf
1960   FOR J = 1 TO fields
```

```
1970      IF VAL F$(J) =  I THEN  pikname$(I) =  N$(J) :
          piklen(I) =  FL(J) : pikpnt(I) = FP(J)
1980      NEXT J
1990    NEXT I
2000 ENDPROC
```

Program 8.4

K Add `PROCselect_fields` to your program.

Sort specification

The sort procedure, `PROCsort`, offers the user three options:

- ☐ Sort on key field in ascending order.
- ☐ Sort on key field in descending order.
- ☐ Leave records unsorted.

The key field is now `pikname$(1)` and not `F$(1)` (unless the user has specifically chosen to use the original key field). The program assumes that the file is too big to sort in memory so it uses the sort method from Program 7.11 of Unit 7.

SAQ 6

Write `PROCsort`. It should:

- ☐ Offer the user the choice of sorting in ascending order, sorting in descending order, or not sorting at all.
- ☐ Load the key field into memory.
- ☐ Sort the key field and `sorter( )`.

Use the array `sort$( )` to store the key field in memory.

- ☐ Add `PROCsort` to your program (your version or ours). You can't test it properly until you have the print procedure.

The sort procedure which you have just written is the limiting part of the program when in practical use. It limits the program in two ways. First, although we have dimensioned `sort$(500)` and `sorter(500)`, even in the Model B Micro there will only be enough memory for the sort if the items in the key field are short. You may therefore prefer to set `maxrecords` (line 80) to a much lower figure. If the items are long you can do a sort of the first 7, say, characters of each item by changing line 2170 to:

```
2170 PTR# CH = 500 + (I - 1) * length + selfp(1) : INPUT# CH, F$
:  sort$(I) = LEFT$(F$, 7)
```

but one way or another, there is a limit to the Micro's sort capacity. Second, the bubble sort which we have used is slow. It is no bother up to 50-60 items but at 100-200 it becomes more tedious. As mentioned, faster sort methods exist but are not covered in this book.

Print procedure

By this stage, PROCselect_data has completed all the selection and sorting specified by the user but he can't see any of it. To see the result the required fields must be printed out in the order of sorter( ). If the user did not request a sort in PROCsort, the print out has to start by initialising sorter( ) to 1, 2, 3, ..., number. The print-out procedure is very simple. It initialises sorter( ) if needed, tells the user to switch on the printer and then reads off the records and prints them. If you do not have a printer attached to your micro then omit 'PROCprinter_on'.

```
2370 DEF PROCprint_out
2380 REM If 'no sort', put sorter( ) into numerical
     order
2390 IF NOT sort THEN FOR I = 1 TO number : sorter(I) = I
     : NEXT I
2400 CH = OPENIN (file$)
2410
2420 REPEAT
2425   PROCprinter_on
2430
2440   FOR record = 1 TO number
2450     FOR I = 1 TO pikf
2460       PTR# CH = 500 + (sorter(record) - 1) * length
           + pikpnt(I)
2470       INPUT# CH, F$(I)
2480       PRINT pikname$(I) TAB(10) F$(I)
2490       NEXT
2500     PRINT
2510     NEXT
2515   CLS : VDU 3 : REM Printer off
2520    PRINT TAB(0,  10) "Press  space bar  to repeat
        print out."  ' "Press ESCAPE for menu."
2530   REPEAT : UNTIL GET$ = " "
2540
2550   UNTIL FALSE
2560 CLOSE# CH
2570 ENDPROC
2580
2590 DEF PROCprinter_on
2600 CLS : PRINT  TAB(0, 10)  "Switch on  printer. Press
     space bar  to start print."
2610 REPEAT : UNTIL GET$ = " "
2615 REM 1200 baud and RS432 output
2620 *FX 5, 2
2625 *FX 8, 4
2630 VDU 2
2640 ENDPROC
```

Program 8.5

Test runs
Using menu Option 3, we can access the data in the file TEST which we created earlier in the unit. Here are two examples.

First run
Print out the records as they stand but sorted on the key field (name).

```
NAME      Andrews
ADDRESS   Mill House, Portsmouth
P. CODE   P12 K44

NAME      Cecil
ADDRESS   Rose Cottage, Norwich
P. CODE   NR6 12HR

NAME      Dearden
ADDRESS   9 Rally Road, Birmingham
P. CODE   B8 9TG

NAME      Jones
ADDRESS   89 Redwood Road, Leeds
P. CODE   L34 5TR

NAME      King
ADDRESS   156 Long Lane, Blackpool
P.CODE    B6 P31

NAME      Lansbury
ADDRESS   Manor House, Cardiff
P. CODE   C23 V55

NAME      Morris
ADDRESS   60 Jubilee Road, Exeter
P. CODE   EX7 J66

NAME      Oliver
ADDRESS   23 Highbury Avenue, Bath
P. CODE   B54 K98

NAME      Williams
ADDRESS   2 Willis Road, Edinburgh
P. CODE   ED4 S62
```

Second run
Print out the records with the post code as the key field, sorted on the key field.

```
P. CODE  B54 K98
NAME     Smith
ADDRESS  23 Highbury Avenue, Bath

P. CODE  B6 P31
NAME     King
ADDRESS  156 Long Lane, Blackpool

P. CODE  B8 9TG
NAME     Dearden
ADDRESS  9 Rally Road, Birmingham

P. CODE  C23 V55
NAME     Lansbury
ADDRESS  Manor House, Cardiff

P. CODE  C8 F5
NAME     Oliver
ADDRESS  6 Cornwall Road, Bingley

P. CODE  ED4 S62
NAME     Williams
ADDRESS  2 Willis Road, Edinburgh

P. CODE  EX7 J66
NAME     Morris
ADDRESS  60 Jubilee Road, Exeter

P. CODE  L34 5TR
NAME     Jones
ADDRESS  89 Redwood Road, Leeds

P. CODE  NR6 12HR
NAME     Cecil
ADDRESS  Rose Cottage, Norwich

P. CODE  P12 K44
NAME     Andrews
ADDRESS  Mill House, Portsmouth
```

Summary

Here is the full program.

```
10 REM File management program
20
30 REM Creates files to users specification
```

```
 40 REM Allows the updating, selecting and sorting of files
 50
 60 MODE 7
 70 maxfields = 20
 80 maxrecords = 500
 90 N = 2 * maxfields
100
110 DIM  H1(N), H2(N),  L(N), F$(N),  V1(N), V2(N),  T$(N),
    N$(maxfields), FL(maxfields), FP(maxfields), pikpnt
    (maxfields), piklen(maxfields), pikname$(maxfields),
    sort$(maxrecords), sorter(maxrecords)
120
130 ON ERROR GOTO 140
140 REPEAT
150
160   REM Menu
170   CLOSE #0
180   CLS
190   PRINT TAB(8, 2) "FILE MANAGEMENT PROGRAM"
200   PRINT TAB(5, 5) "1 Create a new file"
210   PRINT TAB(5) "2 Use a file"
200   PRINT TAB(5) "3 Select data"
230   PRINT TAB(5) "9 End run"
240
250   REM Select option
260   PRINT TAB(0,22) ; "Enter number " ;
270   INPUT "" X
280   IF X = 1 THEN PROCdefine_file : PROCcreate_file
290   IF X = 2 THEN PROCread_header : PROCamend_record
300   IF X = 3 THEN PROCread_header : PROCselect_data
310   IF X = 9 THEN PROCexit
320
330   UNTIL FALSE
340
350 DEF PROCdefine_file
360 LOCAL I, T, ok
370 REM Set up screen data for input
380
390 REPEAT
400   CLS
410   PRINT TAB(0,10) "Enter number of fields required on
      file. " ;
420   INPUT "" fields
430   UNTIL fields >= 1 AND fields <= maxfields
440
450 REM  Set up screen data for field specification
```

```
460 FOR I = 1 TO (maxfields * 2) -1 STEP 2
470   H1(I) = 1 : H2(I) = 13 : V1(I) = 3 + (I + 1) DIV 2 :
      V2(I) = V1(I)
480   H1(I + 1) = 25 : H2(I + 1) = 35 : V1(I + 1) = V1(I) :
      V2(I + 1) = V1(I + 1)
490   T$(I) = "Field " + STR$((I + 1) DIV 2) : T$(I + 1)
      = "Length"
500   L(I) = 8 : L(I + 1) = 3
510   NEXT I
520
530 REM Enter field names and lengths
540
550 REPEAT
560   PROCinput(2 * fields, 1, "Field titles and lengths")
570   ok = TRUE
580   FOR I = 2 TO 40 STEP 2
590     IF VAL(F$(I)) >255 THEN ok = FALSE : F$(I) = "***"
600     NEXT I
610   UNTIL ok
620
630 REM Transfer screen fields to N$( ) and FL( )
640
650 REM  Transfer names to N$( )
660 FOR I = 1 TO 2 * fields - 1 STEP 2
670   N$((I + 1)  DIV 2) = F$(I) : FL((I + 1)  DIV 2) =
      VAL(F$(I + 1))
680   NEXT I
690
700 REM Calculate field pointer positions
710 FOR I = 1 TO fields
720   FP(I) = 0
730   NEXT I
740 FOR I = 2 TO fields
750   FP(I) = FP(I - 1) + FL(I - 1) + 2
760   NEXT I
770
780 REM Calculate record length
790 T = 0
800 FOR I = 1 TO fields
810   T = T + FL(I)
820   NEXT I
830 length = T + 2 * fields
840
850 ENDPROC
860
870 DEF PROCcreate_file
```

```
 880 LOCAL number, I
 890 REPEAT
 900   CLS
 910   PRINT TAB(0, 10) "Enter number of records for
       file (1-"; maxrecords" " ;
 920   INPUT number
 930   UNTIL number >= 1 AND number <= maxrecords
 940
 950 CLS
 960 PRINT TAB(0, 10) "Enter name of file " ;
 970 INPUT "" file$
 980
 990 REM Create fiLe header
1000 CH = OPENOUT (file$)
1010 PRINT# CH, file$, number, length, fields
1020 FOR I = 1 TO fields
1030   PRINT# CH, N$(I), FL(I), FP(I)
1040   NEXT I
1050
1060 REM Create blank records
1070 FOR I = 1 TO number
1080   PTR# CH = 500 + (I - 1) * length
1090   FOR J = 1 TO fields
1100     PTR# CH = 500 + (I - 1) * length + FP(J)
1110     PRINT# CH,""
1120     NEXT J
1130   NEXT I
1140
1150 CLOSE# CH
1160 ENDPROC
1170
1180 DEF PROCamend_record
1190 REPEAT
1200
1210   REPEAT
1220     CLS
1230     PRINT TAB(0, 10) "Enter record number ";
1240     INPUT "" record
1250     UNTIL record >= 1 AND  record <= number
1260
1270   PROCread_write_record(record, FALSE)
1280
1290   REM Set screen data for PROCinput
1300   FOR I = 1 TO fields
1310     H1(I) = 1 : H2(I) = 11 : V1(I) =  3 +  I :  V2(I) =
         V1(I) :  L(I) = FL(I) : T$(I) = N$(I)
```

```
1320     NEXT I
1330
1340   PROCinput(fields, 1, "RECORD ENTRY " + STR$(record))
1350
1360   PROCread_write_record(record, TRUE)
1370
1380   UNTIL FALSE
1390
1400 ENDPROC
1410
1420 DEF PROCselect_data
1430
1440 PROCselect_fields
1450
1460 PROCsort
1470
1480 PROCprint_out
1490
1500 ENDPROC
1510
1520 DEF PROCread_header
1530
1540 REPEAT
1550   CLS
1560   PRINT TAB(0,10) "Enter name of file to be accessed " ;
1570   INPUT "" file$
1580   CH = OPENIN (file$)
1590   UNTIL CH
1600
1610 INPUT# CH, file$, number, length, fields
1620 FOR I = 1 TO fields
1630   INPUT# CH, N$(I), FL(I), FP(I)
1640   NEXT
1650 CLOSE# CH
1660 ENDPROC
1670
1680 DEF PROCread_write_record(record, Z)
1690 CH = OPENIN (file$)
1700 FOR I = 1 TO fields
1710   PTR  #CH = 500 + (record - 1) * length + FP(I)
1720   IF Z = FALSE THEN INPUT# CH, F$(I)
1730   IF Z = TRUE THEN PRINT# CH, F$(I)
1740   NEXT I
1750 CLOSE# CH
1760
1770 ENDPROC
```

```
1780
1790 DEF PROCselect_fields
1800
1810 REM Set up data for screen
1820 FOR I = 1 TO fields
1830   H1(I) = 0 : H2(I) = 12 : V1(I) = 2 + I : V2(I) = V1(I) :
       T$(I) = N$(I) : L(I) = 1 : F$(I) = ""
1840   NEXT I
1850
1860 PROCinput(fields, 1, "SELECT FIELDS (Enter 1,2,3 to select)")
1870
1880 REM Calculate the number of fields chosen (pikf)
1890 pikf = 0
1900 FOR I = 1 TO fields
1910   IF VAL F$(I) >= 1 AND  VAL F$(I) <= fields THEN
       pikf = pikf + 1
1920   NEXT
1930
1940 REM Find names of chosen fields (pikname$( )), lengths
     (piklen(  )) and pointer positions (pikpnt( ))
1950 FOR I = 1 TO pikf
1960   FOR J = 1 TO fields
1970     IF VAL F$(J) =  I THEN  pikname$(I) =  N$(J) :  piklen(I)
         =  FL(J) : pikpnt(i) = FP(J)
1980     NEXT J
1990   NEXT I
2000 ENDPROC
2010
2020 DEF PROCsort
2030
2040 REPEAT
2050   CLS : PRINT TAB(0,  10) "Do  you want  the records
       sorted on  the key field?"
2060   PRINT TAB(0, 13) "Enter:"  ' "N  for no."  ' "A to
       sort  in ascending order." ' "R to sort in reverse order."
2070   INPUT "" R$
2080   IF R$ = "N" THEN sort = FALSE ELSE sort = TRUE
2090   UNTIL LEN R$ = 1 AND INSTR("NAR", R$) <> 0
2100
2110 IF NOT sort THEN ENDPROC
2120
2130 REM Load key sort field into sort$( )
2140 REM ready to sort the key field in memory
2150 CH = OPENIN(file$)
2160 FOR I = 1 TO number
```

```
2170   PTR# CH = 500 + (I - 1) * length + pikpnt(1) :
       INPUT# CH, sort$(I)
2180   NEXT I
2190
2200 REM Sort key field and sorter( )
2210 REM Sorter( ) will be used later to list out file
     in order of sorter( )
2220 FOR I = 1 TO number
2230   sorter(I) = I
2240   NEXT I
2250 FOR K = 1 TO number - 1
2260   FOR L = K + 1 TO number
2270       IF R$  = "A"  THEN IF  sort$(L) <  sort$(K) THEN
           T$ =  sort$(L) : sort$(L) = sort$(K) : sort$(K) =
           T$ : T = sorter(L) : sorter(L) =  sorter(K) :
           sorter(K) = T
2280       IF R$  = "R"  THEN IF  sort$(L) >  sort$(K) THEN
           T$ =  sort$(L) : sort$(L) = sort$(K) : sort$(K) =
           T$ : T = sorter(L) : sorter(L) =  sorter(K) :
           sorter(K) = T
2310     NEXT L
2320   NEXT K
2330 CLOSE# CH
2340
2350 ENDPROC
2360
2370 DEF PROCprint_out
2380 REM If 'no sort', put sorter( ) into numerical order
2390 IF NOT sort THEN FOR I = 1 TO number : sorter(I)
     = I : NEXT I
2400 CH = OPENIN (file$)
2410
2420 REPEAT
2430
2440   FOR record = 1 TO number
2450     FOR I = 1 TO pikf
2460       PTR# CH = 500 + (sorter(record) - 1) *
           length + pikpnt(I)
2470       INPUT# CH, F$(I)
2480       PRINT pikname$(I) TAB(10) F$(I)
2490       NEXT
2500     PRINT
2510     NEXT
2520   PRINT TAB(11) "Press space bar to repeat print  out." '
       "Press ESCAPE for menu."
2530   REPEAT : UNTIL GET$ = " "
```

```
2540
2550   UNTIL FALSE
2560 CLOSE# CH
2570 ENDPROC
2580
2590 DEF PROCprinter_on
2600 CLS : PRINT  TAB(0, 10)  "Switch on  printer. Press
     space bar  to start print."
2610 REPEAT : UNTIL GET$ = " "
2615 REM 1200 baud and RS432 output
2620 *FX 5, 2
2625 *FX 8, 4
2630 VDU 2
2640 ENDPROC
```

Program 8.6

Suggestions for extending the program

1. The program prints out in one format only. Try creating a procedure which allows the user some choice of print format.

2. Because all the fields are string fields, a sort on a field of numbers does not work properly. Modify the sort procedure to allow the user to specify that his sort field is to be sorted as a numeric field.

3. Quite often we want to search a data-base for records which contain a given word, e.g. print out all records containing the word '`KING`'. Write a search procedure to do this (use `INSTR`).

More powerful data bases

Although you can extend this program, there is a limit to how manipulatable you can make random access files. For a more powerful data-base system you need to use a much more complicated file structure such as the Indexed Sequential Access Method (ISAM). If you want to create files of this kind, you should look at one of the specialist works on data-bases, e.g. *Principles of Data Base Management* by Martin James (Prentice-Hall, Inc.)

Assignment H

Either

1. Code any two of the three suggested extensions to Program 8.6.

or

2. Devise an extension of your own to the program.

Objectives of Unit 8

At the end of this unit you should be able to:

- ☐ Design a simple form of random access file structure which allows all files with that structure to be accessed by one program.
- ☐ Apply top-down design to the development of a series of facilities for handling such files.

Answers to SAQs

SAQ 1

The following strings need to be placed in the file header in this order:

Variable name	*String*
file$	STAFF
number	50
length	73 (= FP(4) + FL(4) + 2)
fields	50
N$(1)	Name
FL(1)	25
FP(1)	0
N$(2)	Job
FL(2)	20
FP(2)	27
N$(3)	Dept
FL(3)	15
FP(3)	49
N$(4)	Employee Number
FL(4)	5
FP(4)	66

SAQ 2

```
870 DEF PROCcreate_file
880 LOCAL number, I
890 REPEAT
900   CLS
910   PRINT TAB(0, 10) "Enter number of records for file
     (1-"; maxrecords" " ;
920   INPUT number
930   UNTIL number >= 1 AND number <= maxrecords
940
950 CLS
960 PRINT TAB(0, 10) "Enter name of file " ;
970 INPUT "" file$
```

```
 980
 990 REM Create file header
1000 CH = OPENOUT (file$)
1010 PRINT# CH, file$, number, length, fields
1020 FOR I = 1 TO fields
1030   PRINT# CH, N$(I), FL(I), FP(I)
1040   NEXT I
1050
1060 REM Create blank records
1070 FOR I = 1 TO number
1080   PTR# CH = 500 + (I - 1) * length
1090   FOR J = 1 TO fields
1100     PTR# CH = 500 + (I - 1) * length + FP(J)
1110     PRINT# CH,""
1120     NEXT J
1130   NEXT I
1140
1150 CLOSE# CH
1160 ENDPROC
```

Program 8.7

SAQ 3

```
1520 DEF PROCread_header
1530
1540 REPEAT
1550   CLS
1560   PRINT TAB(0,10) "Enter name of file to be accessed " ;
1570   INPUT "" file$
1580   CH = OPENIN (file$)
1590   UNTIL CH
1600
1610 INPUT# CH, file$, number, length, fields
1620 FOR I = 1 TO fields
1630   INPUT# CH, N$(I), FL(I), FP(I)
1640   NEXT
1650 CLOSE# CH
1660 ENDPROC
```

Program 8.8

Note the crash-proofing loop at 1540 to 1590.

SAQ 4

```
1680 DEF PROCread_write_record(record, Z)
1690 CH = OPENIN (file$)
1700 FOR I = 1 TO fields
```

```
1710   PTR# CH = 500 + (record - 1) * length + FP(I)
1720   IF Z = FALSE THEN INPUT# CH, F$(I)
1730   IF Z = TRUE THEN PRINT# CH, F$(I)
1740   NEXT I
1750 CLOSE# CH
1760
1770 ENDPROC
```

Program 8.9

SAQ 5

```
pikf = 2 (2 fields selected).

pikname$(1) = "Dept"
pikname$(2) = "Name"
piklen(1) = 10
piklen(2) = 20
pikpnt(1) = 22
pikpnt(2) = 0
```

SAQ 6

```
2020 DEF PROCsort
2030
2040 REPEAT
2050   CLS : PRINT TAB(0,  10) "Do  you want  the records
       sorted on  the key field?"
2060   PRINT TAB(0, 13) "Enter:"  ' "N  for no."  ' "A to
       sort  in ascending order." ' "R to sort in reverse order."
2070   INPUT "" R$
2080   IF R$ = "N" THEN sort = FALSE ELSE sort = TRUE
2090   UNTIL LEN R$ = 1 AND INSTR("NAR", R$) <> 0
2100
2110 IF NOT sort THEN ENDPROC
2120
2130 REM Load key sort field into sort$( )
2140 REM ready to sort the key field in memory
2150 CH = OPENIN(file$)
2160 FOR I = 1 TO number
2170   PTR# CH = 500 + (I - 1) * length + pikpnt(1) :
       INPUT# CH, sort$(I)
2180   NEXT I
2190
2200 REM Sort key field and sorter( )
2210 REM Sorter( ) will be used later to list out file
     in order of sorter( )
2220 FOR I = 1 TO number
```

```
2230   sorter(I) = I
2240   NEXT I
2250 FOR K = 1 TO number - 1
2260   FOR L = K + 1 TO number
2270       IF R$  = "A"  THEN IF  sort$(L) <  sort$(K) THEN
           T$ =  sort$(L) : sort$(L) = sort$(K) : sort$(K) =
           T$ : T = sorter(L) : sorter(L) =  sorter(K) :
           sorter(K) = T
2280       IF R$  = "R"  THEN IF  sort$(L) >  sort$(K) THEN
           T$ =  sort$(L) : sort$(L) = sort$(K) : sort$(K) =
           T$ : T = sorter(L) : sorter(L) =  sorter(K) :
           sorter(K) = T
2310     NEXT L
2320   NEXT K
2330 CLOSE# CH
2340
2350 ENDPROC
```

Program 8.10

UNIT 9

GAMES

9.1 Introduction

9.2 Reaction timer

9.3 A simple card game

9.4 A more complicated game

9.5 Movement and collisions

9.6 A game with flight

Assignment I

Objectives of Unit 9

Answers to SAQs

9.1 Introduction

Games are a varied area of computing just as they are in everyday life. However, there are essentially three types of game:

- ☐ games of chance
- ☐ games requiring strategy
- ☐ games of skill.

Before looking at individual games, let's first consider the characteristics of each type of game.

Games of chance
Many simple games are games of pure chance, e.g. snakes and ladders or ludo. In these games there is nothing the player can do to increase his chances of winning. By and large such games are not popular except with very young children who are unaware that the game is controlled by pure chance. For this reason we have not included a game of pure chance in this unit.

Games requiring strategy
The vast majority of games are of this type. Except in the very simplest cases, the number of possible games or moves is so great that no one could ever list them all, nor can a perfect winning strategy necessarily be found. This is what makes chess or bridge so popular. It is also what makes them so difficult to program.

Games of skill
These require the player to be good at some task in which effectively he competes against his own previous score. He may compete against others as well, but this is not essential. Darts, archery or swimming the 100 metres are games of skill. The game which we feature in this category is tiddlywinks, where judging the right combination of velocity and angle are the skills involved. Admittedly, we have written tiddlywinks for two players, but you can play it by yourself and take both parts.

As you will see, even an apparently simple game is far from easy to program. You may therefore consider that the effort required to program games is disproportionate to the rewards. Or you may become hooked on games and be fully prepared to tackle their enormous challenge. At least we hope that this unit will give you enough of a feel of what is involved to help you to decide whether or not you want to delve further into this specialist area of programming.

We start the unit with a very simple game of skill: a reaction timer.

9.2 Reaction timer

Reaction timers are always fun and can, therefore, be included as a type

of game program. More important, the reaction timer is simple enough for us to demonstrate some useful techniques for programs with which naive users interact.

The reaction timer program

The program structure is shown in the flowchart in Figure 9.1.

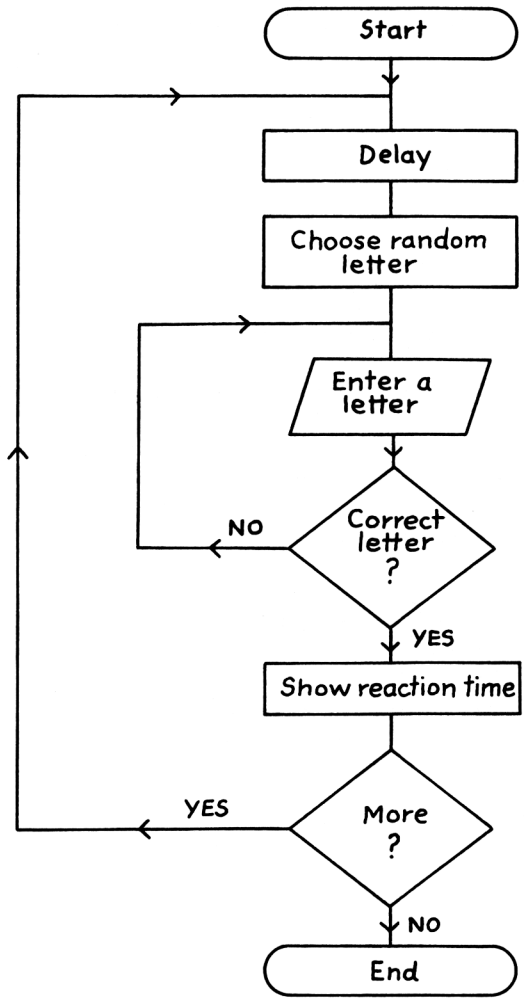


Figure 9.1 Reaction timer flowchart

First, the program chooses a letter at random but we don't bother to represent it as a letter – we just keep it as a number, C (line 170). Secondly, we want to find out how long it takes the user to find this and

press the right key for the chosen letter. We do this with the simple loop:

```
REPEAT
  X = GET
  UNTIL X = C
```

Third, we have used `X = GET` to test whether or not the user wants to have another go. Apart from these points, the program (Program 9.1) is very straightforward.

```
 10 REM Reaction timer
 20
 30 MODE 7
 40
 50 ON ERROR GOTO 60
 60 REPEAT
 70
 80   CLS
 90   PRINT TAB(12, 3) "REACTION TIMER"
100   PRINT TAB(6, 5) "Watch for a letter to appear."
110   PRINT TAB(5, 6) "Press its key as fast as you can."
120
130   REM Make player wait for start
140   PROCdelay
150
160   REM Choose a letter to show
170   C = RND(26) + 64
180
190   REM Wait for player to press key
200   PROCreaction_time
210
220   PRINT TAB(3, 21) "Press the space bar for another go."
230   PRINT TAB(7, 22) "Any other key ends the game."
250
260   UNTIL GET$ <> " "
270
280 CLS
290 END
300
310 REM Procedures
320
330 DEF PROCdelay
340 D = RND(700)
350 t = TIME
360 REPEAT
370   UNTIL TIME >= t + D
```

```
380 ENDPROC
390
400 DEF PROCreaction_time
410 PRINT TAB(19, 10) CHR$(C)
420 start = TIME
430 REPEAT
440   X = GET
450   UNTIL X = C
460 reaction = TIME - start
470 PRINT TAB(2, 20) "Your reaction time was " ;
    reaction / 100 " seconds"
480 ENDPROC
```

Program 9.1

K Type and run Program 9.1.

9.3 A simple card game

The next game which we are going to program is Blackjack.

The rules for our version of Blackjack are:

- There are two players: player and dealer.
- Card values are 2 to 10 as face value; Jack, Queen and King are all 10; Ace is 11.
- Each person tries to get a hand of 21 or as near as possible, but no more than 21.
- Each person is dealt two cards to start with and, if neither has a total of 21, play proceeds as follows:
- Dealer has one card face down, one face up.
- Player's turn: he can take cards one at a time, sticking whenever he likes. If he exceeds 21, he loses. If he gets 21, he wins. If he sticks, play passes to the dealer.
- Dealer's turn. To win, the dealer must now either get 21 or stick at a score above the player's score.

The flowchart

The flowchart is shown in Figure 9.2. As you will see throughout this unit, we keep the programs easy to follow by the use of a main loop controlling the flow of the game. All routines take place outside of the loop, in procedures.

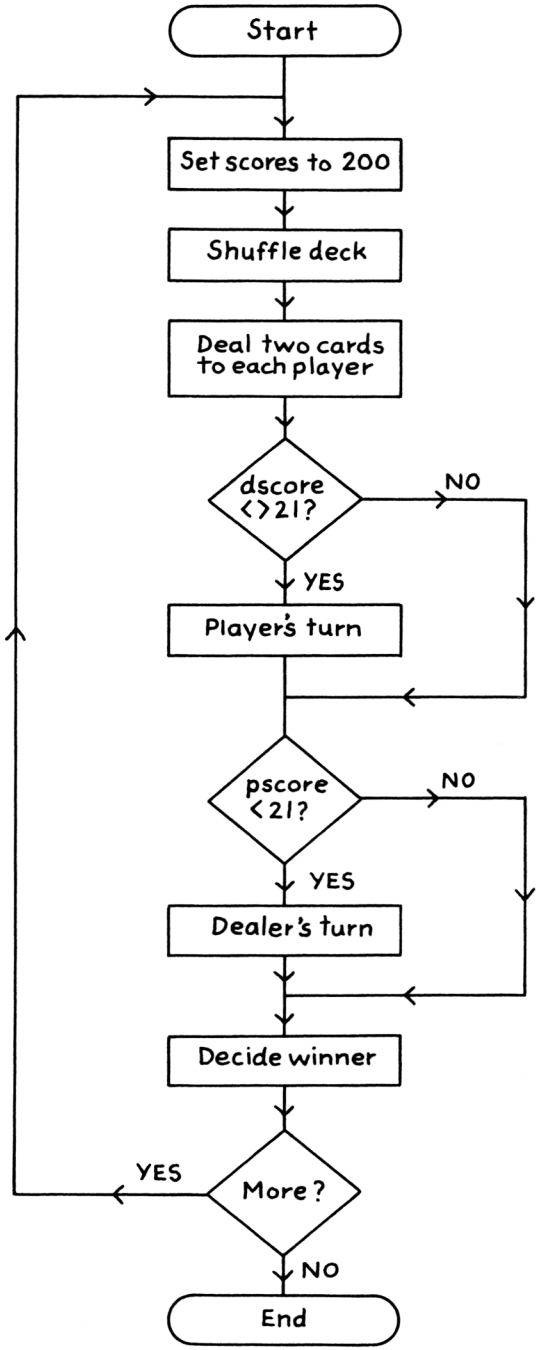


Figure 9.2 Flowchart for Blackjack

The program

The program is build around six procedures:

```
PROCbuild_deck
PROCshuffle
PROCdeal_two_each
PROCplayer
PROCdealer
PROCselect_winner
```

and its structure is clearly shown in the starting section and main loop:

```
 10 REM Blackjack
 20
 30 MODE 7
 40 DIM deck(52)
 50 ON ERROR GOTO 130
 60
 70 REM Re-set total scores
 80 playertotal = 0
 90 dealertotal = 0
100 PROCbuild_deck
110
120 REM Start of main loop
130 REPEAT
140
150   CLS
160   pscore = 0
170   dscore = 0
180
190   PROCshuffle
200
210   PROCdeal_two_each
220
230   IF dscore <> 21 THEN PROCplayer
240   IF pscore < 21 THEN PROCdealer
250
260   PROCselect_winner
270
280   INPUT "Another game? (Y/N) " R$
290
300   UNTIL R$ <> "Y"
310 END
320 REM End of main loop
```

Program 9.2 part a

The main variables are:

```
deck( )     = array of 52 cards
pscore      = player's score
dscore      = dealer's score
nextcard    = position in 'deck( )' of next card to be drawn.
```

The rest of the program is built up, procedure by procedure as follows:

`PROC build_deck` (This is only used once in a run and is therefore called *before* the main loop). The deck is simply an array 'deck( )' containing all possible card values read from the DATA statements. (We don't need to keep track of suits in Blackjack).

```
340 REM Build a deck of cards
350 DEF PROCbuild_deck
360 LOCAL I%
370 FOR I% = 1 TO 52 STEP 4
380   READ deck(I%)
390   FOR J% = 1 TO 3
400     deck(I% + J%) = deck(I%)
410     NEXT J%
420   NEXT I%
430 ENDPROC
...
...
980 REM List of card values for deck
990 DATA 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11
```

Program 9.2 part b

`PROCshuffle` This simply takes 'deck( )' and swaps each item with a randomly selected other item in the deck. (This is just one of many ways in which a computer can simulate the shuffle of a pack of cards).

```
450 REM Shuffle the deck
460 DEF PROCshuffle
470 LOCAL C%, S%, T%
480 FOR C% = 1 TO 52
490   S% = RND(52)
500   T% = deck(C%) : deck(C%) = deck(S%) : deck(S%) = T%
510   NEXT C%
520 ENDPROC
```

Program 9.2 part c

`PROCdeal_two_each` gives each player two cards to start the game. It also displays the player's score and the dealer's first card.

```
540 DEF PROCdeal_two_each
550 REM Start game with two cards each
560 pscore = deck(1) + deck(3)
```

```
570 dscore = deck(2) + deck(4)
580 nextcard = 5
590 PRINT TAB(11, 1) "B L A C K J A C K"
600 PRINT TAB(0, 4) "Player's total    " ; pscore
610 PRINT "Dealer's hand     " ; deck(2)  " + another card"
620 ENDPROC
```

Program 9.2 part d

PROCplayer handles the player's turn and allows him to stick or choose one or more cards.

```
640 DEF PROCplayer
650 REM Player takes card(s) or sticks
660 PRINT ' "PLAYER'S TURN"
670 REPEAT
680   INPUT "Another card? (Y/N) " R$
690    IF  R$ =  "Y" THEN  pscore =  pscore +  deck(nextcard)
       :  nextcard = nextcard + 1 : PRINT "Player's
       total    " ; pscore
700   UNTIL pscore >= 21 OR R$ <> "Y"
710 ENDPROC
```

Program 9.2 part e

PROCdealer handles the dealer's turn and allows him to stick or choose a card(s).

```
730 REM Dealer takes card(s) or sticks
740 DEF PROCdealer
750 LOCAL stick
760 REPEAT
770   stick = FALSE
780   PRINT ' "DEALER'S TURN"
790   PRINT "My total          " ; dscore
800   IF dscore > 17 THEN stick = TRUE : PRINT "I stick."
810    IF NOT  stick THEN  PRINT "I'll  take a  card."
       :  dscore =  dscore + deck(nextcard) : nextcard
       = nextcard + 1 : PRINT "My total                " ;
       dscore
820   UNTIL dscore >= 21  OR stick = TRUE
830 ENDPROC
```

Program 9.2 part f

PROCselect_winner decides the outcome of the game.

```
850 DEF PROCselect_winner
860 REM 21 is winning score
870 scored = FALSE
```

```
880 IF dscore > 21 AND pscore <=  21 THEN  playertotal =
    playertotal +  1 : PRINT "Player wins" : scored = TRUE
890 IF NOT scored THEN IF pscore > 21 AND  dscore <=  21
    THEN  dealertotal = dealertotal + 1  : PRINT "Dealer
    wins" : scored = TRUE
900 IF NOT scored THEN IF pscore <= 21 AND pscore > dscore
    THEN playertotal = playertotal + 1 : PRINT "Player wins"
    : scored = TRUE
910 IF NOT scored THEN IF dscore <= 21 AND dscore > pscore
    THEN dealertotal = dealertotal + 1 : PRINT "Dealer
    wins" : scored = TRUE
920
930 IF NOT scored THEN PRINT "Game drawn"
940 PRINT "My wins            " ; dealertotal
950 PRINT "Your wins          " ; playertotal
960 ENDPROC
```

Program 9.2 part g

K Type and run Program 9.2.

Summary of Blackjack program

```
 10 REM Blackjack
 20
 30 MODE 7
 40 DIM deck(52)
 50 ON ERROR GOTO 130
 60
 70 REM Re-set total scores
 80 playertotal = 0
 90 dealertotal = 0
100 PROCbuild_deck
110
120 REM Start of main loop
130 REPEAT
140
150   CLS
160   pscore = 0
170   dscore = 0
180
190   PROCshuffle
200
210   PROCdeal_two_each
220
230   IF dscore <> 21 THEN PROCplayer
240   IF pscore < 21 THEN PROCdealer
250
```

```
260   PROCselect_winner
270
280   INPUT "Another game? (Y/N) " R$
290
300   UNTIL R$ <> "Y"
310 END
320 REM End of main loop
330
340 REM Build a deck of cards
350 DEF PROCbuild_deck
360 LOCAL I%
370 FOR I% = 1 TO 52 STEP 4
380   READ deck(I%)
390   FOR J% = 1 TO 3
400     deck(I% + J%) = deck(I%)
410     NEXT J%
420   NEXT I%
430 ENDPROC
440
450 REM Shuffle the deck
460 DEF PROCshuffle
470 LOCAL C%, S%, T%
480 FOR C% = 1 TO 52
490   S% = RND(52)
500   T% = deck(C%) : deck(C%) = deck(S%) : deck(S%) = T%
510   NEXT C%
520 ENDPROC
530
540 DEF PROCdeal_two_each
550 REM Start game with two cards each
560 pscore = deck(1) + deck(3)
570 dscore = deck(2) + deck(4)
580 nextcard = 5
590 PRINT TAB(11, 1) "B L A C K J A C K"
600 PRINT TAB(0, 4) "Player's total    " ; pscore
610 PRINT "Dealer's hand     " ; deck(2)  " + another card"
620 ENDPROC
630
640 DEF PROCplayer
650 REM Player takes card(s) or sticks
660 PRINT ' "PLAYER'S TURN"
670 REPEAT
680   INPUT "Another card? (Y/N) " R$
690    IF  R$ =  "Y" THEN  pscore =  pscore +  deck(nextcard) :
       nextcard = nextcard + 1 : PRINT "Player's total    " ; pscore
```

```
700   UNTIL pscore >= 21 OR R$ <> "Y"
710 ENDPROC
720
730 REM Dealer takes card(s) or sticks
740 DEF PROCdealer
750 LOCAL stick
760 REPEAT
770   stick = FALSE
780   PRINT ' "DEALER'S TURN"
790   PRINT "My total          " ; dscore
800   IF dscore > 17 THEN stick = TRUE : PRINT "I stick."
810    IF NOT  stick THEN  PRINT "I'll  take a  card." :
       dscore =  dscore + deck(nextcard) : nextcard = nextcard+1
       : PRINT "My total            " ; dscore
820   UNTIL dscore >= 21  OR stick = TRUE
830 ENDPROC
840
850 DEF PROCselect_winner
860 REM 21 is winning score
870 scored = FALSE
880 IF dscore > 21 AND pscore <=  21 THEN  playertotal =
    playertotal +  1 : PRINT "Player wins" : scored = TRUE
890 IF NOT scored THEN IF pscore > 21 AND  dscore <=  21
    THEN  dealertotal = dealertotal + 1  : PRINT "Dealer wins"
    : scored = TRUE
900 IF NOT scored THEN IF pscore <= 21 AND pscore > dscore
    THEN playertotal = playertotal + 1 : PRINT "Player wins"
    : scored = TRUE
910 IF NOT scored THEN IF dscore <= 21 AND dscore > pscore
    THEN dealertotal = dealertotal + 1 : PRINT "Dealer wins"
    : scored = TRUE
920
930 IF NOT scored THEN PRINT "Game drawn"
940 PRINT "My wins           " ; dealertotal
950 PRINT "Your wins         " ; playertotal
960 ENDPROC
970
980 REM List of card values for deck
990 DATA 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11
```

Program 9.2

9.4 A more complicated game

Blackjack has introduced the essential ideas of games where player and computer compete against each other. It was fairly easy to program for two reasons: (a) there were only two moves – player followed by

computer, after which the result is announced, and (b) the computer was given a very simple strategy to play by. We shall now look at a very simple game (on paper) which is much more difficult to program. The game is noughts and crosses. As you will see, the programming problems arise from the fact that there is no simple set of rules by which a computer can play noughts and crosses. Even an apparently simple set of rules results in some complex decision making in the program.

The strategy

Before we can program a game with the computer as one player, we must be able to formulate a strategy for it to follow. The more complicated you make the strategy, the more complicated the program becomes. So, in the interests of keeping the problem manageable, we shall get the computer to follow this strategy (computer plays 'X'):

Computer's first move

- ☐ Take the centre if it is free, otherwise take a corner, chosen at random.

Computer's later moves

- ☐ Look for a row with 'XX' in it and no 'O's'. If such a row exists, fill the gap with 'X'. Computer wins.
- ☐ If no 'XX' row exists, look for a row with 'OO' in it and no 'X's, i.e. a row in which the player can win on his next move. If such a row exists, fill the gap with an 'X'.
- ☐ If no 'XX' or 'OO' row exists, look for a row with 'X' and no 'O's and place an 'X' in one of the empty squares.

As you will see, this strategy is moderately effective. The last rule is the weak one since there may be more than one 'X' row and, in any 'X' row, there are two blank spaces to choose from. Ideally the computer should look for the best 'X' to play but we shall ignore that refinement to the game.

The board and the variables

- ☐ We shall number the board

7	8	9
4	5	6
1	2	3

for convenience from the point of view of the computer graphics. (If the display were not graphical, other numberings might be more convenient).

The main variables:

`player`	Decides whose move: `player = TRUE` is player's move, `player = FALSE` is computer's move.
`moves`	Number of moves so far (`moves` counts up to 9)
`win`	Result state: `win = 1`: computer wins `win = -1`: player wins `win = 0`: draw
`B$(n)`	Is the state of the nth position in the board. All `B$( )` locations are `""` at the start and become `'X'` or `'O'` as they are filled.
`T$`	Shows the state of the paths through the board, e.g. the state of path 1, 5, 9 is found from `T$ = B$(1) + B$(5) + B$(9)`. So `T$` has values such as `'X'`, `'XO'`, `'XXX'` etc.

The paths

There are 8 paths through the board. These are stored in DATA lines so that the paths can be checked before the computer moves (to decide its move) and after both computer and player moves (to look for a win).

The flow diagram

- ☐ The main loop alternates between moves and the display of the moves as `I` increments from 1 to 9. According to whether player is `TRUE` or `FALSE`, the computer offers play to itself or the player. `player` is changed at the end of the loop.
- ☐ After each pass round the loop, the computer checks (line 340) to see whether any of the 'end' conditions have been reached, i.e. `moves = 9` or `win = 1` or `win = -1`. If one of these conditions is met, the computer exits from the loop and prints the appropriate message (lines 360 - 400).
- ☐ The computer has two types of move: first move (to take the centre, or if occupied, a corner) or later moves (look for `'XX'` and win, look for `'OO'` and block player, look for `'X'` and play another `X`).

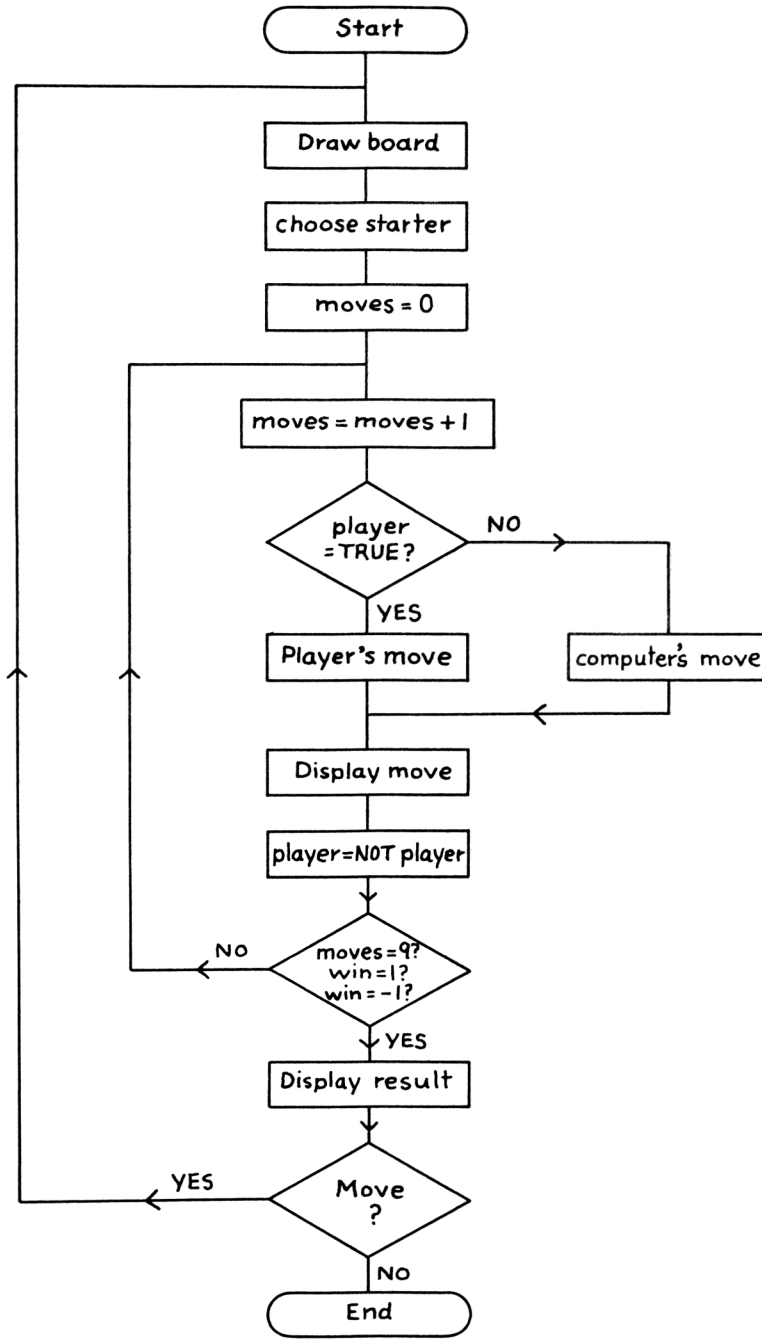


Figure 9.3 Noughts and crosses flowchart

The procedures

PROCplayers_move
This uses PROCsearch ('OO') to check for a win.

PROCcomputer_move
There are four types of move for the computer:
- ☐ A first move where the computer tries to take the centre of the board.
- ☐ A search for a line with 'XX'.
- ☐ A search for a line with 'OO'.
- ☐ A search for a line with 'X'.

Having found the type of path it wants, it uses PROCfill_gap (F,G,H) to find the gap in the path and put an "X" in it.

PROCboard
This creates the board on the screen.

PROCdisplay_move
Places the latest 'X' or 'O' on the board.

PROCsearch
PROCsearch is sent a string when called and then searches each path on the board for that string. F, G and H take values from the DATA list, e.g. 1, 2, 2 (first path) or 9, 5, 1 (last path). T$ = B$(F) + B$(G) + B$(H) establishes what is on that path. Then T$ is compared with the required string, S$. The PROC is left when either T$ = S$ or there are no more paths to check. Note the word 'RESTORE' which ensures that the DATA list is read from the beginning – RESTORE is essential when a program is to use DATA more than once.

PROCfill_gap
When PROCsearch has found a path on which to put the 'X', PROCfill_gap is called to find where the gap is (at B$(F), B$(G) or B$(H)) and to put an 'X' in the gap. If there is more than one gap, then the first gap to be found is filled. P is set to the number of the square that was filled ready for displaying the new 'X'.

```
10 REM Noughts and crosses
20
30 DIM B$(9)
40 MODE 5
50
60 REM Text window
70 VDU 28, 1, 31, 18, 29
80
90 REPEAT
100
110   PROCboard
120
130   INPUT "Who starts? (P/C) " R$
```

```
140   IF R$ = "P"  THEN player = TRUE ELSE player = FALSE
150
160   REM Initialise variables for next game
170   win = FALSE
180   moves = 0
190   T$ = ""
200   FOR P = 1 TO 9
210     B$(P) = ""
220     NEXT P
230
240   REPEAT
250
260     moves = moves + 1
270
280     IF player THEN PROCplayers_move ELSE PROCcomputer_move
290
300     PROCdisplay_move
310
320     player = NOT player
330
340     UNTIL moves = 9 OR win
350
360   IF win = 1 THEN PRINT "I win."
370   IF win = -1 THEN PRINT "You win."
380   IF win = 0 THEN PRINT "Draw."
390
400   INPUT "Another game? Y/N " G$
410   UNTIL G$ <> "Y"
420 CLS
430 CLG
440 END
450
460 REM End of main loop
470
480 DEF PROCcomputer_move
490 IF moves <= 2 THEN PROCcomputer_move_1
500 IF moves > 2 THEN PROCcomputer_later_move
510 ENDPROC
520
530 DEF PROCcomputer_move_1
540 moved = FALSE
550 IF moves = 1 THEN B$(5) = "X" : P = 5 : moved = TRUE
560 IF moves = 2 AND B$(5) = ""  THEN B$(5) = "X" : P = 5 :
    moved = TRUE
570 IF NOT moved THEN P = VAL(MID$("1279", 1 + RND(3), 1))
```

```
580 IF NOT moved THEN B$(P) = "X"
590 ENDPROC
600
610 DEF PROCplayers_move
620 REPEAT
630
640   REPEAT
650     INPUT "Your move. " P
660     IF P < 1 OR P > 9 THEN PRINT "Invalid move"
670     UNTIL P >= 1 AND P <= 9
680
690   IF B$(P) <> ""  THEN PRINT "Square already taken."
700   UNTIL B$(P) = ""
710 B$(P) = "O"
720
730 REM Check for player win
740 PROCsearch("OOO")
750 IF T$ = "OOO" THEN win = -1
760 ENDPROC
770
780 DEF PROCcomputer_later_move
790 filled = FALSE
800
810 REM Look for winning move
820 PROCsearch("XX")
830 IF T$ = "XX" THEN PROCfill_gap(F, G, H) : REM Computer's
    winning move
840 IF filled THEN win = 1
850
860 REM Need to block player?
870 IF NOT filled THEN PROCsearch("OO")
880 IF NOT filled THEN IF T$ = "OO" THEN PROCfill_gap(F, G,  H)
    :  REM Block player's move
890
900 REM Computer's offensive move
910 IF NOT filled THEN PROCsearch("X")
920 IF NOT filled THEN PROCfill_gap(F, G, H)
930 ENDPROC
940
950 DEF PROCboard
960 CLG
970 MOVE 0, 900
980 GCOL 0, 2
990 VDU 5 : PRINT "NOUGHTS AND CROSSES" : VDU 4
1000 MOVE 150, 200
1010 DRAW 150, 800
```

```
1020 DRAW 1050,800
1030 PLOT 85,150,200
1040 DRAW 1050,200
1050 DRAW 1050,800
1060 PLOT 85,150,200
1070 GCOL 0, 1
1080 FOR X = 0 TO 3
1090   FOR Y = 0 TO 3
1100     MOVE 150 + X * 300, 200
1110     DRAW 150 + X * 300, 800
1120     MOVE 150,200+Y*200
1130     DRAW 1050, 200 + Y * 200
1140     NEXT Y
1150   NEXT X
1160 VDU 5
1170 FOR P = 1 TO 9
1180   MOVE ((P - 1) MOD 3) * 300 + 160, ((P - 1) DIV 3)
       * 200 + 235
1190   PRINT ; P
1200   NEXT P
1210 VDU 4
1220 ENDPROC
1230
1240 DEF PROCdisplay_move
1250 MOVE ((P - 1) MOD 3) * 300 + 300, ((P - 1) DIV 3)
     * 200 + 300
1260 VDU 5 : PRINT B$(P) : VDU 4
1270 ENDPROC
1280
1330 REM Search board for X/O combinations
1340 DEF PROCsearch(S$)
1350 RESTORE
1360 C = 0
1370 REPEAT
1380   READ F, G, H
1390   C = C + 1
1400   T$ = B$(F) + B$(G) + B$(H)
1410   UNTIL T$ = S$ OR C = 8
1420 ENDPROC
1430
1440 DEF PROCfill_gap(F, G, H)
1450 IF B$(F) = "" THEN P = F ELSE IF B$(G) = "" THEN P
     = G ELSE P = H
1460 B$(P) = "X"
1470 filled = TRUE
1480 ENDPROC
```

```
1490
1500 REM Data for board paths
1510 DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 7, 4, 1
1520 DATA 8, 5, 2, 9, 6, 3, 7, 5, 3, 9, 5, 1
```

Program 9.3

K Type and run Program 9.3.

Discussion

This program demonstrates several things about programming games. First, even a comparatively simple game like noughts and crosses is complex when you wish to reduce it to yes/no questions required for a program. Second, a 'reasonable' solution will usually be found through adopting a strategy which ensures that the computer wins most of the time. To aim to get it always to win will usually be too difficult and quite pointless – who would play it? Third, to keep track of what is happening, it is essential to use procedures whenever possible.

Extensions

If you want to try to improve the program, some possibilities are:

- ☐ Arrange for the choice of who starts the first game to be a random decision with the start of games thereafter alternating between players.
- ☐ Build in a scoring system.
- ☐ The weak part of the strategy is lines 900 to 920 where the computer finds it ought to consider all such paths, and choose the one most likely to promote a computer win. You could get the computer to compute a score for all the possible moves and then to choose the move that maximises its own score or its own chance of a high score.

9.5 Movement and collisions

We have already seen movement on the screen in the clock program (Unit 4). Let's take a closer look at the process. The basic principle involves printing a character (or whatever) at successive locations on the screen and wiping out (unplotting) its previous position. This results in movement across the screen.

SAQ 1

Write a program to move an asterisk up the screen from 500, 0 to 500, 1000.

Varying the speed

You may want fast or slow movement. Almost certainly you never want the speed that you get when you try a program for the first time! Let's look first at slowing movement down, since that is very simple to achieve.

SAQ 2

Modify Program 9.9 (SAQ 1 answer) to make the `STEP` value a variable `speed`, and add an input line to allow you to run the program with different values of `speed`.

You will quickly have found that small values of `speed` slow the movement down. Here then is an effective way of making your character (or whatever) move more slowly.

But what about speeding up the movement? There are various techniques available for speeding up the loop without resorting to assembly language programming. These are:

(a) Remove any unnecessary calculations from the loop and do these once only before entry to the loop.

(b) Use integer variables in place of floating point variables where possible. i.e. if `I` is always a whole number, call it `I%` instead.

(c) Put more statements on one line. It looks a mess but does speed things up.

Here is the asterisk printing program speeded up using (b) and (c) – it doesn't offer any opportunity for using (a).

```
10 MODE 5 : VDU 5
20 FOR I% = 1 TO 1000 STEP 10 : GCOL0, 1 : MOVE 500, I% :
   PRINT " * " : GCOL0, 0 : MOVE 500, I% : PRINT " * " : NEXT
```

Program 9.4

Collisions

So far, we have made the asterisk move to a fixed point (500, 1000) but we often find that we want to move an object until a condition is met e.g.

move until the edge of the board
move until hitting a starship
move until falling in the lake.

This should immediately suggest that the movement will be controlled by `REPEAT ... UNTIL ...`

SAQ 3

Rewrite Program 9.10 (answer to SAQ 2) using `REPEAT ... UNTIL ...`

The essential point about collision with another object is that the collision may or may not take place. If it doesn't take place, the asterisk continues on its path. Movement takes place

```
UNTIL (hit_starship OR off_screen).
```

Example 1
Write a program to place a starship ("S") at a random position on the route of the asterisk. The asterisk and the starship are to disappear on collision.

Solution

```
10 REM Collisions
20
30 MODE 5
40 VDU 5
45 REM Set random position for starship
50 in = 200 + RND(800)
60 up = RND(1000)
70
80 REM Print starship
90 GCOL 0, 2
100 MOVE in, up
110 PRINT "S"
120
130 REM Fire asterisk
140 I = 10
150 REPEAT
160
170   I = I + 10
180   GCOL 0, 2
190   MOVE in, I
200   PRINT "*"
210   GCOL 0, 0
220   MOVE in, I
230   PRINT "*"
240
250   UNTIL I >= up
260
270 REM Remove starship and asterisk
280 GCOL 0, 0
290 MOVE in, I
300 PRINT "*"
310 MOVE in, up
320 PRINT "S"
```

Program 9.5

The method used in Program 9.5 will only work when the stepsize of the moving starship is less than the size of the target. When this is not the case it is possible for the starship to jump over the target without a collision.

K Type and run Program 9.5.

The final type of 'accident' that might befall a moving object is to meet a larger area, e.g. an area of screen that represents, say, a lake or a hole. There are two ways of testing whether an object has fallen in the lake or the hole. One is to work out whatever condition is needed to define the relevant area of the screen. The other is to use POINT.

POINT

POINT (X, Y) tells you the number of the colour at the position (X, Y) on the screen. (If the position is off the screen, POINT (X, Y) = -1). The following program places a number of blue lakes in random locations across the path of the asterisk. If the asterisk falls into a lake, it disappears, otherwise it travels on to the top of the screen. Without POINT, we would have to create a complex test to find out whether or not the asterisk had fallen into one of the lakes.

```
 10 REM Detecting colour at a point
 20
 30 MODE 5
 40 VDU 5
 50 VDU 19, 3, 4, 0, 0, 0
 60 in = 500
 70
 80 PROClakes(20, 200)
 90
100 REM Move asterisk through terrain
110 J = 10
120 REPEAT
130
140   J = J + 10
150   GCOL 0, 1
160   MOVE in, J
170   PRINT "*"
180   GCOL 0, 0
190   MOVE in, J
200   PRINT "*"
210
220   UNTIL J >= 1000 OR POINT(in, J) = 3
230
240 IF POINT(in, J) <> 3  THEN END
250
260 REM Drown asterisk in blue
270 MOVE in, J
280 GCOL 0, 3
290 PRINT "*"
300 END
310
320 DEF PROClakes(lakenum, size)
```

```
330 FOR I = 1 TO lakenum
340   X = RND(1000) : Y = RND(1000)
350   GCOL 0, 3
360   MOVE X, Y
370   DRAW X + size, Y
380   DRAW X + size, Y + size
390   PLOT 85, X, Y
400   DRAW X, Y + size
410   DRAW X + size, Y + size
420   PLOT 85, X, Y
430   NEXT I
440 ENDPROC
```

Program 9.6

K Type and run Program 9.6.

Now that we have explained the general principles of movement, you can go on to more complex examples using two advanced techniques that are explained in the *User Guide*. These are (a) creating your own character shapes (*User Guide* pp 170 - 172) and (b) creating shapes larger than the size of one print character (*User Guide* pp 172 - 175).

Two-dimensional movement

Two-dimensional is a little more complex, but still quite manageable.

First, let's modify Program 9.6 to produce two dimensional movement in a straight line. Put a new line in at line 135:

```
135 in = in + 5
```

Run the program, then try other values than 5 in line 135. Try `in = in - 5`.

Second, let's look at how curved two dimensional paths can be created. The example that we are going to look at involves the flight of a projectile (e.g. a cricket ball) as it moves under the pull of gravity.

Any object launched by a simple initial push into the air traces out a parabola as it rises and then falls back to the ground (Figure 9.4).

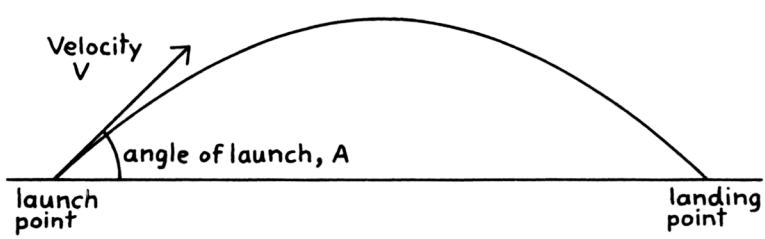


Figure 9.4 A parabolic flight path

The height reached by the object and its distance of travel are determined by, amongst other variables, the velocity V and the angle A. (We shall ignore things such as air resistance and wind speed). If we want to draw its flight on the screen, we need to know how far it has travelled and how high it is at various times during its flight. After T seconds, the position is given by:

Horizontal distance travelled = V * T * COS(A)
Height from the ground = T * (V * SIN(A) - 16 * T)

where V = velocity at start
T = time since start
A = angle of launch

Don't worry about the maths or where these formulae come from! You can use them whether or not you understand sine and cosines. All we have to do is to move the '*' to the position given by the formulae. To make the program easier to follow, we have made:

```
X = V * COS(A)
Y = Y * SIN(A)
```

The '*' is printed at successive positions found by incrementing T by .1 each time. Line 310 introduces a short delay between each draw operation.

```
 10 REM Flight demonstration
 20
 30 MODE 7
 40
 50 REM Input flight details
 60 PRINT TAB(0, 10) ;
 70 INPUT "Velocity        " V
 80 INPUT "Angle (degrees) " D
 90 REM Convert angle to radians
100 A = RAD D : REM Convert angle to radians
110
120 REM Flight display
130 MODE 5
140 VDU 5
150 T = 0
160 REM Vertical scale five times horizontal scale
170 REM to give better display
180 X = V * COS(A)
190 Y = 5 * V * SIN(A)
200
210 REPEAT
```

```
220   T = T + .1
230   along = T * X
240   up = T * (Y - T * 16)
250   MOVE along, up
260   GCOL 0, 1
270   PRINT "*"
280   MOVE along, up
290   GCOL 0, 0
300   PRINT "*"
310   K = INKEY 2
320   UNTIL up <= 0
330
340 VDU 4
350 END
```

Program 9.7

K Type and run Program 9.7.

This program produces an acceptable version of flight despite the fact that in real life T varies continuously and not in 0.1 steps as in our model.

9.6 A game with flight

Using our flight algorithm, we are going to write a program for a game of tiddlywinks between two players, neither of whom is the computer. The computer simply calculates and displays the game as played by the players.

Unlike the other games we have met so far, tiddlywinks (at least as we shall play it) presents no great problems of strategy. Nor does it require complicated decision-making routines. Instead, tiddlywinks shows another aspect of game construction: the need to produce effective graphics and to keep them under control. Most of the program is graphics, but graphics of things that move. You won't find our program perfect and you may wish to take up the challenge of making it 'look' even better.

The game model

The game is restricted to two dimensions with a 'board' (Figure 9.5). Each player has three counters and the first player to get all three counters in the pot wins. (A counter is simply a text '1' or '2'). Play proceeds as follows:

- One counter is placed on the screen for each player in a random position. Player 1's counter is yellow, player 2's is red.
- They take turns to shoot by entering a velocity (1 to 250) and an angle (1 to 89).

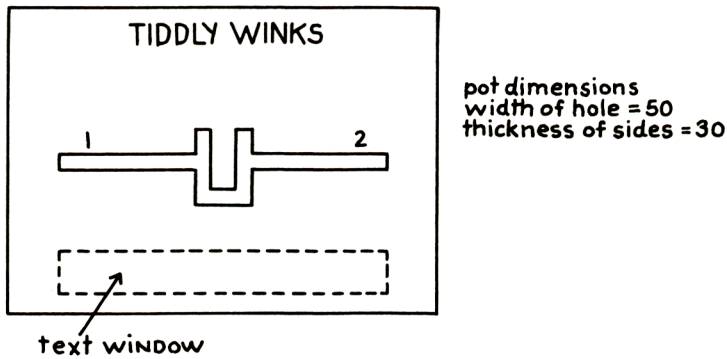


Figure 9.5 Tiddlywinks board

- ☐ The counter flies across the screen to its new position unless (a) it hits the outside of the pot or (b) it goes outside the area of play. In either of these events, the counter is bounced back to a new random position.
- ☐ When a player gets a counter in the pot, a new counter is placed on the board for him.

The flowchart for the game is shown in Figure 9.6.

The program structure

Superficially the program looks complex but we have structured it to make it easy to follow or to amend. Every major routine is done in a procedure whilst the program flow is controlled by three nested `REPEAT ... UNTIL ...` loops. (These can be seen as the three loops on the left-hand side of the flow chart.) Program flow is controlled by the conditions at the end of the loops.

- ☐ Is it on the board or in the pot? (line 350)
- ☐ Has one of the players won? (line 400)
- ☐ Do they want another game? (line 480)

The program uses `MOD` (line 240) and `ABS` (lines 330 to 350). In line 240 the purpose of `MOD` is to create a function (`P MOD 2 + 1`) which, if `P` has the value 1 changes it to 2 and, if `P` has the value 2, changes it to 1. In this way we can alternate play between player 1 and player 2. `ABS` is used to make the program treat negative distances (distances left of the pot) as if they were positive. `ABS(-20)` is 20, just as is `ABS(20)`.

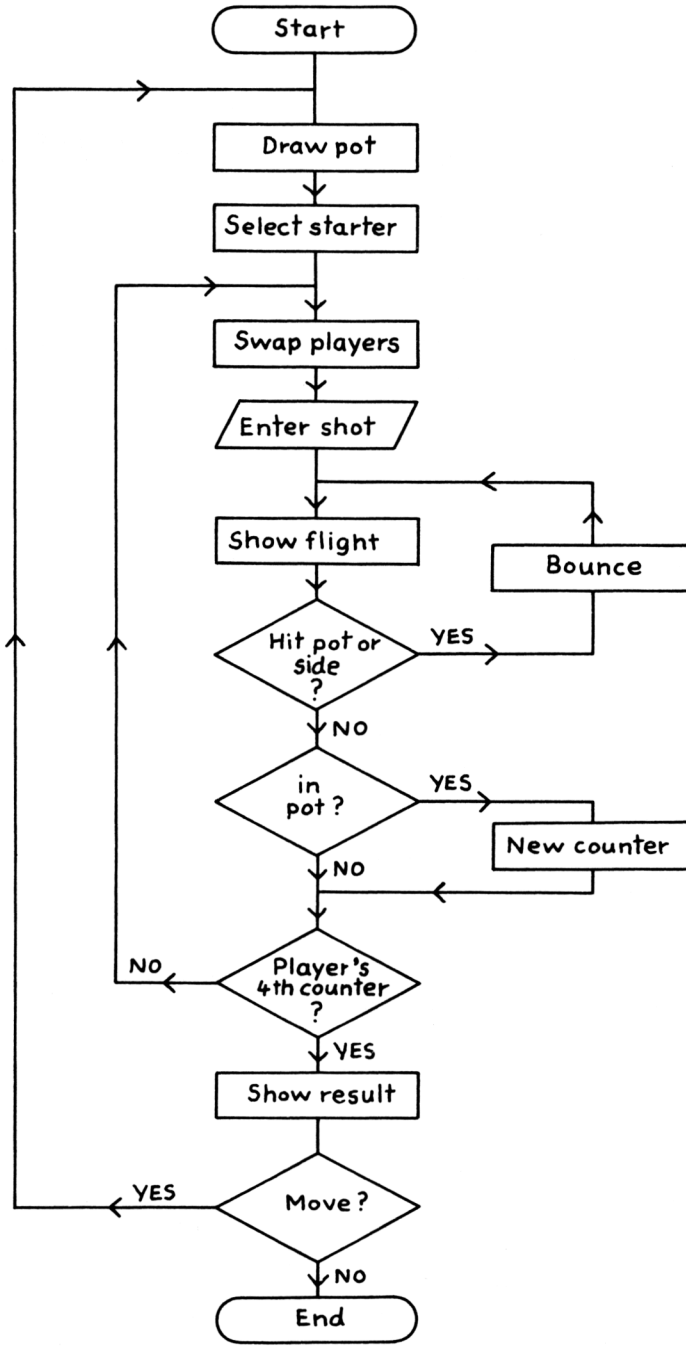


Figure 9.6 Tiddlywinks flowchart

```
 10 REM Tiddlywinks
 20
 30 MODE 5
 40 DIM counter(2), along(2), up(2)
 50 width = 50 : REM width of hole in pot
 60 margin = 30 : REM notional thickness of edge of pot
 70 max = 450 : REM notional edge of play area
 80
 90 REM P = player number
100 REM counter(P) = counter No. in play
110 REM along(P) = distance player P is from pot
120 REM up(P) = height of counter P from board
130
140 REPEAT
150
160   REM  Set up pot
170   PROCpot
180   P = RND(2)
190
200   REM Main loop
210   REPEAT
220
230     REM Alternate players (swaps P = 1 to P = 2 and vice versa)
240     P = (P MOD 2) + 1
250
260     PROCinput_shot
270
280     REM Flight path on screen
290     REPEAT
300       PROCflight
310       REM If hits pot or side, bounce off
320       REM then calculate new path
330         IF ABS  along(P) >  width AND  along(P) <=  width +
            margin THEN velocity =  RND(250) : angle = RND(45) :
            PROCflight
340       IF ABS along(P) > max THEN velocity = RND(250) : angle
          = RND(45) : PROCflight
350        UNTIL  ABS along(P)  <=   max AND  along(P) >  width +
           margin OR along(P) <= width
360
370     REM Counter in pot?
380     IF ABS along(P) <= width THEN PROCnew_counter
390
400     UNTIL counter(1) = 4 OR counter(2) = 4
410
420   REM Win
```

```
430   CLS
440   REM Flashing colour for win message
450   VDU 19, P+1, 9, 0, 0, 0
460   COLOUR P + 1
470   PRINT "PLAYER " ; P " WINS"
480   INPUT "ANOTHER GAME? Y/N " R$
490   REM Turn off flashing colour
500   VDU 19, P + 1, 7, 0, 0, 0
510
520   UNTIL R$ <> "Y"
530 END
540 REM End of main loop
```

Program 9.8 part a

SAQ 4

We now want you to build the program, module by module. Don't worry if your modules are not exactly the same as ours as long as they follow the standards of design and coding that we've used in this book. Write the program in the following stages:

PROCpot
Write this to:

- ☐ Create a text window near the bottom of the screen.
- ☐ Move the origin to 600, 500.
- ☐ Draw the pot (as shown in Figure 9.5).
- ☐ Place two counters at random positions along the board 100 points above the board. (Make sure the counters are not too near the pot.)

⌨ Test your `PROCpot` and compare it with ours.

SAQ 5

PROCinput_shot
Write this to:

- ☐ Allow the player to input his velocity (`V`) and angle of shot (`A`).

⌨ Type in your `PROCinput_shot` and compare it with ours.

PROCflight
This uses the method we showed in Program 9.5. The steps are:

- ☐ Unplot start position.
- ☐ Make a small movement.
- ☐ Show new position.
- ☐ Unplot at new position.

Then, on exit from the loop, the counter is printed at its new position.

'Make a small movement' is best found by trial and error – we have incremented T (time) by .25 for each step. This produces a reasonably smooth flight at a slow enough speed across the screen to give the game some suspense. Here is the flight routine:

```
820 DEF PROCflight
830 PROCprint_counter(P, 0, along(P), up(P))
840 IF along(P) > 0 THEN K = -1 ELSE K = 1
850 U = K * velocity * (COS RAD(angle))
855 W = velocity * SIN RAD(angle)
860
870 FOR T = 0 TO W / 16 STEP .25
880   Y = up(P) + T * (W - 16 * T)
890   X = along(P) + U * T
900   PROCprint_counter(P, P+1, X, Y)
910   PROCprint_counter(P, 0, X, Y)
920   NEXT T
930
940 along(P) = along(P) + U * (T - .25)
950 up(P) = 100 + (T - .25) * (W - 16 * (T - .25))
960 PROCprint_counter(P, P + 1, along(P), up(P))
970 ENDPROC
```

Program 9.8 part b

K Run your program with `PROCflight` added. To get the program to run, you will need to put in a temporary short cut for `PROCnew_counter`, i.e. add

```
DEF PROCnew_counter
ENDPROC
```

so that you can test the program whilst this is missing.

SAQ 6

Once `PROCflight` is running in your program, try writing `PROCnew_counter` for yourself. This needs to:

- ☐ Rub out the counter.
- ☐ Put it in the pot.
- ☐ Put a new counter on the board.

K Test your `PROCnew_counter` and compare it with ours.

The full program is as follows:

```
 10 REM Tiddlywinks
 20
 30 MODE 5
 40 DIM counter(2), along(2), up(2)
 50 width = 50 : REM width of hole in pot
 60 margin = 30 : REM notional thickness of edge of pot
 70 max = 450 : REM notional edge of play area
 80
 90 REM P = player number
100 REM counter(P) = counter No. in play
110 REM along(P) = distance player P is from pot
120 REM up(P) = height of counter P from board
130
140 REPEAT
150
160   REM  Set up pot
170   PROCpot
180   P = RND(2)
190
200   REM Main loop
210   REPEAT
220
230     REM Alternate players
240     P = (P MOD 2) + 1
250
260     PROCinput_shot
270
280     REM Flight path on screen
290     REPEAT
300       PROCflight
310       REM If hits pot or side, bounce off
320       REM then calculate new path
330         IF ABS  along(P) >  width AND  along(P) <=  width +
            margin THEN velocity =  RND(250) : angle = RND(45)
            : PROCflight
340       IF ABS along(P) > max THEN velocity = RND(250) :
          angle = RND(45) : PROCflight
350        UNTIL  ABS along(P)  <=   max AND  along(P) >
           width +  margin OR along(P) <= width
360
370     REM Counter in pot?
380     IF ABS along(P) <= width THEN PROCnew_counter
390
400     UNTIL counter(1) = 4 OR counter(2) = 4
410
420   REM Win
```

```
430   CLS
440   REM Flashing colour for win message
450   VDU 19, P+1, 9, 0, 0, 0
460   COLOUR P + 1
470   PRINT "PLAYER " ; P ; " WINS"
480   INPUT "ANOTHER GAME? Y/N " R$
490   REM Turn off flashing colour
500   VDU 19, P + 1, 7, 0, 0, 0
510
520   UNTIL R$ <> "Y"
530 END
540 REM End of main loop
550
560 DEF PROCinput_shot
570 CLS
580 PRINT "Player No. " ; P
590
600 REPEAT
610   INPUT "Velocity (1-250) " velocity
620   UNTIL velocity >= 1 AND velocity <= 250
630
640 REPEAT
650   INPUT "Angle (1-89 degrees) " angle
660   UNTIL angle >= 1 AND angle <= 89
670
680 ENDPROC
690
700 DEF PROCnew_counter
710 REM Wipe out potted counter
720 PROCprint_counter(P, 0, along(P), up(P))
730 REM and show it in pot
740 PROCprint_counter(P, P + 1, RND(40) - 20, RND(100) - 150)
750 REM Get new counter
760 counter(P) = counter(P) + 1
770 IF counter(P) = 4 THEN ENDPROC
780 along(P) = 200 + RND(400)
785 up(P) = 100
790 PROCprint_counter(P, P + 1, along(P), up(P))
800 ENDPROC
810
820 DEF PROCflight
830 PROCprint_counter(P, 0, along(P), up(P))
840 IF along(P) > 0 THEN K = -1 ELSE K = 1
850 U = K * velocity * (COS RAD(angle))
855 W = velocity * SIN RAD(angle)
860
```

```
 870 FOR T = 0 TO W / 16 STEP .25
 880   Y = up(P) + T * (W - 16 * T)
 890   X = along(P) + U * T
 900   PROCprint_counter(P, P+1, X, Y)
 910   PROCprint_counter(P, 0, X, Y)
 920   NEXT T
 930
 940 along(P) = along(P) + U * (T - .25)
 950 up(P) = 100 + (T - .25) * (W - 16 * (T - .25))
 960 PROCprint_counter(P, P + 1, along(P), up(P))
 970 ENDPROC
 980
 990 DEF PROCpot
 995 LOCAL I, P
1000 CLG
1010 VDU 28, 0, 31, 19, 27 : REM Text window
1020 VDU 29, 600; 500; : REM Move origin to (600, 500)
1030 counter(1) = 1
1040 counter(2) = 1
1050 GCOL 0, 1
1060 VDU 5
1070 MOVE -400, 400
1080 PRINT "TIDDLYWINKS"
1090 FOR I = -1  TO 1 STEP 2
1100   MOVE 600 * I, 0 : DRAW 600 * I, 30 : DRAW 50 * I, 30 :
       PLOT 85,  600 * I, 0
1110   DRAW 50 * I , 0 : DRAW 50 * I, 30 : PLOT 85, 600 * I, 0
1120   MOVE 80 * I, 60 : DRAW 80 * I, -200 : DRAW 50 * I, 60 :
       PLOT 85,  80 * I, 60
1130   MOVE 50 * I, -200 : DRAW 50 * I, 60 : PLOT 85,
       80 * I, -200
1140   DRAW -55 * I, -200 : DRAW -55 * I, -170 : PLOT 85,
       55 * I, -200
1150   DRAW 55 * I, -170 : DRAW -55 * I, -170 : PLOT 85,
       55 * I, -200
1160   NEXT I
1170
1180 FOR P = 1 TO 2
1190   along(P) = 200 + RND(400)
1200   up(P) = 100
1210   PROCprint_counter(P, P + 1, along(P), up(P))
1220   GCOL 0, colour
1230   NEXT P
1240
1250 VDU 4
1260 ENDPROC
```

```
1270
1280 DEF PROCprint_counter(counter, colour, along, up)
1290 VDU 5
1300 MOVE along, up
1310 GCOL 0, colour
1320 PRINT ; counter
1330 VDU 4
1340 ENDPROC
```

Program 9.8

K Type and run the full version of Program 9.8.

Assignment I

1. A simple game works as follows. At the bottom of the screen is a 'bat' which the player can move from side to side. A game consists of ten balls, offered to the player one at a time. Each ball is fired off the top of the screen straight down to the bottom. If the ball hits the bat, a point is scored. If it misses the bat, no point is scored. The program should count and display the total number of hits which the player makes with his ten balls. Write the program, making the maximum use of procedures and variables. This will then enable you to write a more complex version of the game later if you want to.

Objectives of Unit 9

When you have finished studying this unit you should be able to write programs for simple games incorporating:

- ☐ Random features.
- ☐ Timing.
- ☐ Decision-making routines for wins.
- ☐ Scoring routines.
- ☐ Strategies for the computer to play.
- ☐ Movement of text or graphics characters.
- ☐ Control of the speed of movement.
- ☐ Collisions by position.
- ☐ Collisions by colour (using `POINT`).

Answers to SAQs

SAQ 1

A possible program is:

```
 10 REM MOVEMENT
 20
 30 MODE 5
 40 VDU 5
 50 FOR I = 1 TO 1000 STEP 10
 60   GCOL 0, 1
 70   MOVE 500, I
 80   PRINT " * "
 90   GCOL 0, 0
100   MOVE 500, I
110   PRINT " * "
120   NEXT I
```

Program 9.9

SAQ 2

```
 10 REM Movement with variable step
 20
 30 MODE 7
 40
 50 PRINT TAB(0, 10);
 60 INPUT "Speed for character move? " speed
 70
 80 MODE 5
 90 VDU 5
100 FOR I = 1 TO 1000 STEP speed
110   GCOL 0, 1
120   MOVE 500, I
130   PRINT "*"
140   GCOL 0, 0
150   MOVE 500, I
160   PRINT "*"
170   NEXT I
```

Program 9.10

K Type and run Program 9.10.

SAQ 3

```
 10 REM Movement controlled by 'UNTIL'
 20
 30 MODE 5
 40 VDU 5
 50
 60 I = 10
 70 REPEAT
 80   I = I + 10
 90   GCOL 0, 1
100   MOVE 500, I
```

```
 101   PRINT "*"
 110   GCOL 0, 0
 120   MOVE 500, I
 121   PRINT "*"
 130   UNTIL I >= 1000
```

Program 9.11

K Type and run Program 9.11.

SAQ 4

```
 990 DEF PROCpot
 995 LOCAL I, P
1000 CLG
1010 VDU 28, 0, 31, 19, 27 : REM Text window
1020 VDU 29, 600; 500; : REM Move origin to (600, 500)
1030 counter(1) = 1
1040 counter(2) = 1
1050 GCOL 0, 1
1060 VDU 5
1070 MOVE -400, 400
1080 PRINT "TIDDLEYWINKS"
1090 FOR I = -1  TO 1 STEP 2
1100   MOVE 600 * I, 0 : DRAW 600 * I, 30 : DRAW 50 * I, 30 :
       PLOT 85,  600 * I, 0
1110   DRAW 50 * I , 0 : DRAW 50 * I, 30 : PLOT 85, 600
       * I, 0
1120   MOVE 80 * I, 60 : DRAW 80 * I, -200 : DRAW 50 * I, 60 :
       PLOT 85,  80 * I, 60
1130   MOVE 50 * I, -200 : DRAW 50 * I, 60 : PLOT 85,
       80 * I, -200
1140   DRAW -55 * I, -200 : DRAW -55 * I, -170 : PLOT 85,
       55 * I, -200
1150   DRAW 55 * I, -170 : DRAW -55 * I, -170 : PLOT 85,
       55 * I, -200
1160   NEXT I
1170
1180 FOR P = 1 TO 2
1190   along(P) = 200 + RND(400)
1200   up(P) = 100
1210   PROCprint_counter(P, P + 1, along(P), up(P))
1220   GCOL 0, colour
1230   NEXT P
1240
1250 VDU 4
1260 ENDPROC
```

Program 9.12

(We have used multi-statement lines for drawing the pot (lines 1100 to 1150) because the coding is essentially trivial).

SAQ 5

```
560 DEF PROCinput_shot
570 CLS
580 PRINT "Player No. " ; P
590
600 REPEAT
610   INPUT "Velocity (1-250) " velocity
620   UNTIL velocity >= 1 AND velocity <= 250
630
640 REPEAT
650   INPUT "Angle (1-89 degrees) " angle
660   UNTIL angle >= 1 AND angle <= 89
670
680 ENDPROC
```

Program 9.13

SAQ 6

```
700 DEF PROCnew_counter
710 REM Wipe out potted counter
720 PROCprint_counter(P, 0, along(P), up(P))
730 REM and show it in pot
740 PROCprint_counter(P, P + 1, RND(40) - 20,
    RND(100) - 150)
750 REM Get new counter
760 counter(P) = counter(P) + 1
770 IF counter(P) = 4 THEN ENDPROC
780 along(P) = 200 + RND(400)
785 up(P) = 100
790 PROCprint_counter(P, P + 1, along(P), up(P))
800 ENDPROC
```

Program 9.14

UNIT 10

Testing and de-bugging

10.1 Types of programming error

10.2 Systematic testing

10.3 Other techniques

10.1 Types of programming error

There are three main types of error which you can make when writing computer programs: syntax errors, run-time (or system) errors and logic errors.

Syntax errors

These are the easiest errors to locate since the computer identifies them for you. A syntax error means trying to write a statement which is an incorrect variation of the programming language that you are using. Trying to execute

```
10 INPUT "Next item" ; item$
```

produces the message 'Syntax error at line 10' since BBC BASIC does not permit ';' as part of an input command.

To locate all of the syntax errors in a program, proceed as follows:

- ☐ Remove all the error traps in your program.
- ☐ Run the program under varying conditions and try to ensure that *all* statements in the program are executed at least once.

This last point needs some explanation. Consider a program segment such as

```
1000 INPUT option
1020 ON option GOTO 2000, 3000, 4000 ELSE ; 10
```

If you test the program as it is meant to be used (i.e. entering 1, 2 or 3 at line 1000), the `ELSE` action is never executed. Yet there lurks a bug (the ";" after `ELSE`) which will be discovered as soon as you type in a number which is greater than 3.

When syntax errors are found, the error message will give you the line number. This is not always the line number where the error lies e.g. an error in procedure arguments can result in a confusing error message since we are told that the error is in the procedure definition line when it is actually in the line that called the procedure. Running the following program produces such an error message:

```
 10 REM Centering text
 20
 30 chars = 80 : REM Mode 1
 40 T$ = "LOOKS BEST IN CENTRE"
 50 line = 10
 60 PROCcentre(T, line)
 70
 80 END
 90
100 DEF PROCcentre(text$, line)
```

```
110 PRINT TAB((chars - LEN text$) DIV 2, line) text$
120 ENDPROC

>RUN

No such variable at line 100
```

Program 10.1

[K] Now change line 60 (not line 100) to `PROCcentre(T$, line)` and run again.

Run-time errors

These are errors which involve syntactically correct statements which cannot be executed by the computer. They arise from the way that the program is written but cannot be seen to be invalid purely by visual inspection of the syntax. Here are some examples:

(a) `100 PRINT average`

will produce the message '`No such variable at line 100`' if 'average' has not previously been assigned a value.

(b)
```
10 DIM N(5)
20
30 FOR I = 1 TO 5
40 READ N(I)
50 NEXT I
60 END
70
80 DATA 233, 45, "WORD", 12, 10
```

Program 10.2

produces the message '`Type mismatch at line 40`' since the DATA line contains a string but the `READ` line only assigns the data to a numeric variable.

(c)
```
10 A = 3E24
20 B = 4E20
30 PRINT A * B
```

Program 10.3

produces the message '`Too big at line 30`' since the product of A and B is larger than 1E38 (the largest value that a numeric variable can take in BBC BASIC).

Run-time errors are much harder to locate than syntax errors since they may not show up on every run. For example, a variable may only exceed the computer's numeric range on certain runs. There are various techniques which can be used to minimise such errors:

- ☐ Restrict input data to the minimum range needed for the program to do its job. If a variable only needs to take values between 0 and 5, use a trap to prevent other values being entered.
- ☐ Test the program over a range of values. Errors often occur with end-of-range values.
- ☐ Ensure that the computer has enough memory for all your variables. To do this proceed as follows:
- ☐ At the start of the program, dimension all arrays and assign values to all numeric variables.
- ☐ Fill the string variables that the program will use with strings of the maximum length that can arise in the program. This not only checks that there is enough memory for the strings but overcomes the 'string garbage' problem that the BBC Micro has. This problem can arise when you assign a new string to a string location. If the new string is longer than the last string of that name, the computer takes a new block of memory for the new string value and discards the old section of memory. It does not re-use the old block during the current run. You can prevent this as in the following example.

A program might use the variables `A`, `B`, `C`, `D%`, `I` and `P$(max length 70)` together with an array of 50 strings each of maximum length 50. Then we can check whether or not enough memory is available by filling all these variables as follows:

```
10 REM Declare variables
20 A = 0 : B = 0 : C = 0 : D% = 0 : I = 0
30
40 REM Dimension arrays
50 DIM array$(50)
60
70 REM Fill array strings
80 FOR I = 0 TO 50
90 array$(I) = STRING$(50, " ")
100 NEXT I
110
120 REM Fill strings
130 P$ = STRING$(70, " ")
```

Program 10.4

If there is not enough memory for the variables, a '`No room`' error will occur. If you don't get an error message, you still can't be certain that the program has enough room to run since a BASIC program needs space for the 'BASIC stack' which holds return addresses for procedures, definitions and loops. The BASIC stack is placed just below **HIMEM** and the amount of space required will depend on the logic of your program.

Logic errors

These are the hardest errors to detect, since the computer can offer no help at all. Logic errors are all of those errors which are not syntax or run-time errors. As far as the computer is concerned, logic errors are not errors at all since the program makes sense to the computer. The trouble is that the program is not solving the problem you meant it to solve but some variation on that problem.

10.2 Systematic testing

Debugging a program can be a search for a bug that has appeared once a program is in use. The aim of testing is to try to prevent that state ever being reached. What we want to do is to prove that a program works correctly before it is ever used. Now we can't quite prove that no bug will ever appear but what we can do is to work in a systematic manner that will take us as near to proof as we can reasonably get. We shall now demonstrate an approach to systematic testing on Program 10.5.

```
 10 REM Find the highest number
 20 REM Use for a list of positive numbers only
 30
 40 REM highest = highest number found so far
 50 REM number = an item of data
 60 REM total = number of data items read
 70
 80 PROCinitialise
 90
100 PROCsearch_highest
110
120 PROCprint_result
130
140 PROCexit
150
160
170 DATA 23, 34, 13, 9, 0, 7, 87, 215, 2, 1, 0, -999999
180
190 DEF PROCinitialise
200 highest = 0
210 number = 0
220 total = 0
230 ENDPROC
240
250 DEF PROCsearch_highest
260 REPEAT
270   READ number
280   IF number < 0 AND number <> -999999 THEN PROCwrong_data
```

```
290   IF number <> -999999 THEN PROCcheck
300   UNTIL number = -999999
310 ENDPROC
320
330 DEF PROCprint_result
340 PRINT "List contains " ; total " numbers."
350 PRINT "Highest number in list is " ; highest
360 ENDPROC
370
380 DEF PROCcheck
390 IF number > highest THEN highest = number
400 total = total + 1
410 ENDPROC
420
430 DEF PROCwrong_data
440 CLS
450 PRINT TAB(0, 10) "Invalid data in list."
460 PROCexit
470 ENDPROC
480
490 DEF PROCexit
500 PRINT "Run ended."
510 END
```

Program 10.5

The steps are:

(a) *List all the variables* and examine their initialisation. Look at each occurrence of the variable and check that the spelling is correct and that the right data type has been used.

Variable	Initialisation OK?	Type	Occurs at	Spelling OK?	Correct type?
highest	Yes(=0)	FP	200	Yes	Yes
			340	Yes	Yes
			390	Yes	Yes
number	Yes(=0)	FP	270	Yes	Yes
			280	Yes	Yes
			290	Yes	Yes
			300	Yes	Yes
			390	Yes	Yes
total	Yes(=0)	FP	220	Yes	Yes
			340	Yes	Yes
			400	Yes	Yes

(b) *Test top-down*

The procedures in the program can be thought of as being at various levels in the top-design according to how they are called:

```
Level 1   initialise        search_highest      print_results    exit

Level 2             wrong_data                        check
```

We first need to check that level 1 of the top-down design is correct. To do this, close off all level 2 procedures by putting an ENDPROC after the first line of their definitions, i.e. add

```
435 ENDPROC : REM !!!!!!
385 ENDPROC : REM !!!!!!
```

(The REM !!!!!! makes it easy to find these lines later when you want to remove them). If there were any level 3 procedures (i.e. ones called by level 2 procedures but not by level 1 procedures) then these would automatically be closed off by the closing off of the level 2 procedures.

Testing level 1

Now run the program, stopping it after each level 1 procedure, i.e.

Insert STOP at line 85.
Run the program.
Check all the variables at line 85.
If all is correct, then PROCinitialise is correct.

Remove line 85.
Insert 105 STOP.
Run the program.
Check all the variables at line 105.
Repeat for different sets of data.
If all is correct, then PROCinitialise and PROCsearch_highest are alright.

Finish testing level 1 in this way.

Testing level 2

Now remove lines 385 and 435 and repeat the level 1 test with STOPs at 85, 105, 125 and 145. If all is correct, then the level 2 procedures work correctly.

Testing level 3

If there were any level 3 procedures, you would now remove the dummy ENDPROCs from them and continue with level 3 testing.

(c) *Vary the data systematically*

The program must be forced to run through a suitable variety of sets of data including silly values, values which it should reject, etc. Where possible, do this by enclosing the level 1 procedures in a loop which runs the program several times and varies the data each time. Let's put Program 10.5 through several runs – four with test data and one with the existing data in line 170.

First, put the loop in by adding the lines:

```
 71 FOR I = 1 TO 5
125 NEXT I
```

Second, add new data lines:

```
171 DATA -999999                          (Test for coping with no
                                           data)
172 DATA 3, 45, 98, -3, 0, -999999        (Test for rejecting a nega-
                                           tive number)
173 DATA 0, 0, 0, 0, -999999              (Test with a lot of zeros)
174 DATA 999999, 887655, 0, 1,            (Some large numbers)
    -999999
```

Third, make the program read the appropriate data for each test run:

```
72 IF I = 1 THEN RESTORE 170
73 IF I = 2 THEN RESTORE 171
74 IF I = 3 THEN RESTORE 172
75 IF I = 4 THEN RESTORE 173
76 IF I = 5 THEN RESTORE 174
```

Fourth, allow the tester to control the five runs:

```
77 PRINT "PRESS SPACE BAR FOR NEXT TEST"
78 REPEAT : UNTIL GET$ = " "
```

When you actually try this test, you will find that the second test (data line 172) forces an exit from the program (as it should). You have therefore to remove line 172 in order to complete the test.

(d) *Vary the data randomly*

To do this requires a bit of messing about with the program itself. We can't put `RND` into the data line in order to place random data there so we have to put `"RND( )"` there and then use `EVAL` after the read operation. So, first change the data line to:

```
170 DATA "RND(1000)", "RND(1000)", "RND(1000)", "RND(1000)",
    "-999999"
```

Then change line 270 to:

```
READ T$
```

And add:

```
IF T$ <> "-999999" THEN number = EVAL(T$) ELSE number = -999999
```

You would also need to get the program to print the random numbers that it had generated in order to check that it had found the highest in every case.

5 Boundary testing

Very often a program will work with random data but produces an incorrect result the moment somebody uses it routinely. This is because everyday use will not be random – the data will have meaning. Often the trouble will lie in boundary values i.e. when data takes values at which a test becomes true. It is therefore important to test a program to make sure that it behaves correctly at its boundary values.

Consider line 280 of Program 10.5. This is to stop the program if a negative number is found in the data. It is therefore essential that the test data should include:

Values just below the boundary	e.g.	–1
Values at the boundary	i.e.	0
Values just above the boundary	e.g.	1

and that it is run to ensure that the correct decision has been made each time. In Program 10.5, the correct decision will be made each time, but test the boundary conditions with this modified line 280:

```
280 IF number <= 0 AND number <> -999999 THEN PROCwrong_data
```

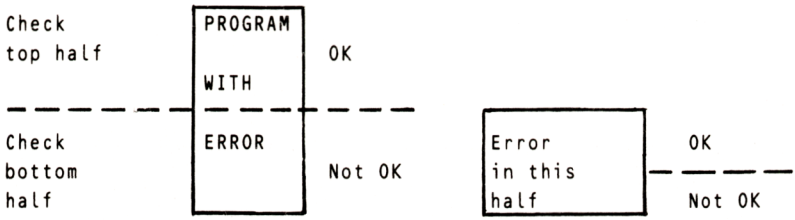
Another common example of a boundary error can occur with array dimensioning e.g.

```
10 DIM A(10)
20
30 FOR I = 0 TO 10
40   A(I) = A(I - 1)                          (Or A(I) = A(I + 1))
50   NEXT I
```

Whichever version of line 40 you use, an error will arise at the boundaries (i.e. when `I = 0` and `I = 10`).

Binary search

Another approach is to use a binary search technique and work from both ends of a program in order to identify where an error is located:



This method is of limited value since it assumes the program has only one line of flow with a single starting point and a single finishing point.

10.3 Other techniques

Keyboard techniques

Whether you test top-down or use a binary search, you will need to use certain techniques in order to find logic errors.

1. Printing out intermediate data

Unless you can see from its output why a program doesn't work, you will need to consider how the program reached that output. To do this you should insert extra print commands into the program to show you what is happening at different stages.

Example 1

Look back at Program 3.9 (the word-print routine). During its insertion into a program, the 'I' in line 170 was accidentally typed as a '1'). The resulting output was bizarre. Where was the error? Well, the routine repeats an operation but we only see the end result. In order to find the error we need to see the steps of the repetition. To do this we insert a new line at line 175:

```
175 PRINT "I = " ; I " word$ = "word$ " P$ = " P$
```

The print out from this showed that `P$` was retaining letters that should have been allocated to `word$`. Inspection of the program should then lead to a closer examination of line 170 and so reveal the error in that line.

2. Checking all program variables

Providing that you are running a program without error traps, you can press Escape at any point in a run in order to check what values the variables have at the point where you stopped the run. After Escape has been pressed, all variables retain their values and can be inspected by displaying them with immediate-mode `PRINT` commands.

Example 2

```
10 REM Just a test program
20
30 DIM A$(20)
```

```
40
50 FOR I = 1 TO 20
60 FOR J = 1 TO 20
70 A$(J) = A$(J) + CHR$(RND(26) + 64)
80 NEXT J
90 NEXT I
```

K ☐ Run this program.

☐ Press Escape during the run.

☐ Then type as follows:
`FOR K = 1 TO 20 : PRINT A$(K) : NEXT` (press Return).

Here is our print out:

```
VJHO
BDGV
XNSO
TGBL
RZV
HJC
GQZ
EWP
IPX
FDF
FKU
EMN
GAN
JOX
HLK
LPD
BRD
QAZ
ZTX
NTC
```

Then type:
`PRINT I, J` (press Return)
Here is our result:

```
              4           5
```

We know that at the moment we pressed Escape, the program was on its fourth pass of the `I` loop and the fifth pass of the `J` loop. The values for `A$( )` were as listed above.

Note: If you have escaped from a program, you can re-start it at any line by typing

```
GOTO (Line number) (press Return)
```

Program execution then re-starts at that line number. If you do this, however, don't try to re-start inside a loop, procedure, defined function or subroutine. You must limit your re-starting to a line which is not associated with any looping or stack activity.

Another method of doing this is to include within your program a special procedure, say, `PROCvarprint`, which lists the values of all the variables. It is never called by the program. Instead, you press Escape and then call `PROCvarprint` in immediate mode.

3. Stepping through values

Let's look at two further techniques for making a program execute in a way which allows you to follow what is happening more easily.

First, you can insert lines at various points in the program which will halt execution until you do something. One simple way of doing this to insert line such as

```
REPEAT : UNTIL GET$ = " "
```

This causes program execution to wait until the space bar is pressed. This is particularly useful as a means of giving you time to study the output from the temporary print lines that you have inserted to give you intermediate values of your data.

Second, you can enclose a program, or section of program, in an extra loop which successively puts the program through a series of initial values for one or more of the program variables.

Example 3

If you were in the process of devising Program 5.3 (the circle drawing procedure), you might wish to establish the minimum number of steps which will produce a decent looking circle. To find this out, you can enclose the procedure in a loop which draws successive circles, clearing the screen after each. When you see the first circle that looks adequate, press Escape and see what value the loop variable has. (Or print it in the top corner of the screen as it runs).

4. Using TRACE

Sometimes a program's behaviour is baffling, e.g. when it hangs without outputting any message. (This usually occurs because the program is in an endless loop). When this is the case, you probably don't know where to begin to look for the error and at this point `TRACE` might help. The statement

```
TRACE ON
```

causes the computer to print out the line number of every line executed by the interpreter.

Example 4
Type and run the following program:

```
10 REM Something wrong
20
30 I = 0
40 REPEAT
50 I = I + 2
60 UNTIL I = 191
70 PRINT "Finished."
```

Press Escape when you are fed up!

What is happening? Why is it taking so long to finish? To see what is going on, type 'TRACE ON' and then run the program again. The screen fills with the line numbers 50 and 60 showing that the computer is repeatedly executing lines 50 and 60. Press Escape to halt the run. You can then examine lines 50 and 60 in order to see what is happening. (In this case you should be able to see that the condition 'UNTIL I = 191' will never be met since I is always even. Perhaps the programmer meant 'UNTIL > = 191').

To get rid of TRACE, type TRACE OFF.

BBC MICRO CHARACTER SET

Code Dec (Hx)	Char	Code Dec (Hx)	Char	Code Dec (Hx)	Char
0 (0)	Does nothing	42 (2A)	*	84 (54)	T
1 (1)	Next character to printer	43 (2B)	+	85 (55)	U
2 (2)	Enable printer	44 (2C)	,	86 (56)	V
3 (3)	Disable printer	45 (2D)	-	87 (57)	W
4 (4)	Separate text/graphics cursors	46 (2E)	.	88 (58)	X
5 (5)	Join text/graphics cursors	47 (2F)	/	89 (59)	Y
6 (6)	Enable VDU drivers	48 (30)	0	90 (5A)	Z
7 (7)	Make a short beep	49 (31)	1	91 (5B)	[
8 (8)	Backspace cursor one character	50 (32)	2	92 (5C)	\
9 (9)	Forwardspace cursor one character	51 (33)	3	93 (5D)	]
10 (A)	Move cursor down one line	52 (34)	4	94 (5E)	↑
11 (B)	Move cursor up one line	53 (35)	5	95 (5F)	_
12 (C)	Clear text area	54 (36)	6	96 (60)	
13 (D)	Move cursor to start of line	55 (37)	7	97 (61)	a
14 (E)	Page mode on	56 (38)	8	98 (62)	b
15 (F)	Page mode off	57 (39)	9	99 (63)	c
16 (10)	Clear graphics area	58 (3A)	:	100 (64)	d
17 (11)	Define text colour	59 (3B)	;	101 (65)	e
18 (12)	Define graphics colour	60 (3C)	<	102 (66)	f
19 (13)	Define logical colour	61 (3D)	=	103 (67)	g
20 (14)	Restore default logical colours	62 (3E)	>	104 (68)	h
21 (15)	Disable VDU drivers	63 (3F)	?	105 (69)	i
22 (16)	Select screen mode	64 (40)	@	106 (6A)	j
23 (17)	Re-program display character	65 (41)	A	107 (6B)	k
24 (18)	Define graphics window	66 (42)	B	108 (6C)	l
25 (19)	Plot m, x, y	67 (43)	C	109 (6D)	m
26 (1A)	Restore default windows	68 (44)	D	110 (6E)	n
27 (1B)	Does nothing	69 (45)	E	111 (6F)	o
28 (1C)	Define text window	70 (46)	F	112 (70)	p
29 (1D)	Define graphics origin	71 (47)	G	113 (71)	q
30 (1E)	Home text cursor to top left	72 (48)	H	114 (72)	r
31 (1F)	Move text cursor to x, y	73 (49)	I	115 (73)	s
32 (20)	Space	74 (4A)	J	116 (74)	t
33 (21)	!	75 (4B)	K	117 (75)	u
34 (22)	"	76 (4C)	L	118 (76)	v
35 (23)	#	77 (4D)	M	119 (77)	w
36 (24)	$	78 (4E)	N	120 (78)	x
37 (25)	%	79 (4F)	O	121 (79)	y
38 (26)	&	80 (50)	P	122 (7A)	z
39 (27)	'	81 (51)	Q	123 (7B)	{

40	(28)	(	82	(52)	R
41	(29)	)	83	(53	S

124	(7C)	|
125	(7D)	}
126	(7E)	~
127	(7F)	Delete

APPENDIX 2

POLAR COORDINATES

The Cartesian system

The commonest method of locating a point on a flat surface is to use Cartesian coordinates. This system uses two axes (usually labelled the x-axis and y- axis). The point at which the axes meet is called the origin (O) or centroid. Any point on the surface is uniquely identified by its distance from the two axes.

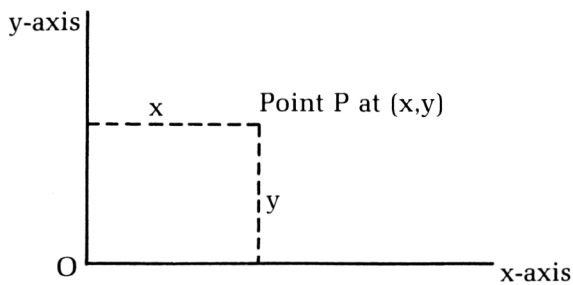


Figure A2.1 The Cartesian system of coordinates

The point P has coordinates (x, y).

Polar coordinates

Polar coordinates perform the same function as Cartesian coordinates, i.e. they uniquely locate a point on a flat surface. However, polar coordinates use a point and a line as their basic grid.

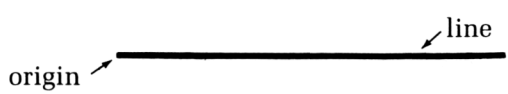


Figure A2.2 The reference point and line of polar coordinates

A point is located in polar coordinates by its distance from the origin and the angle that a line between the origin and the point makes with the initial line. In Figure A2.3, the point P has polar coordinates (5, PI / 6).

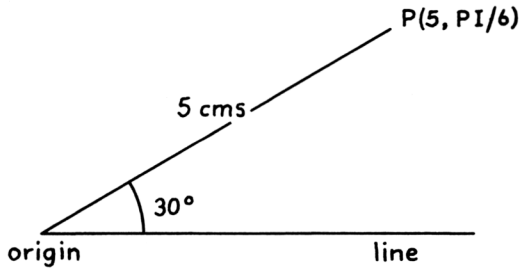


Figure A2.3 The polar coordinates of the point P

Notice that the angle is expressed in radians (PI / 6) and not degrees (30°). You may not have met radians. They are simply an alternative method of measuring an angle. A circle contains 360 degrees or 2 x PI radians. To convert from degrees to radians use

radians = (degrees x PI) / 180

and to convert from radians to degrees use

degrees = (radians x 180) / PI

The conversions in BBC BASIC are even easier. To convert from degrees to radians you use `RAD` and to convert from radians to degrees you use `DEG`:

radians = `RAD` (degrees)
degrees = `DEG` (radians)

Why use polar coordinates?
For some purposes, Cartesian co-ordinates are the easiest to use; for other purposes, polar coordinates are easiest. Polar coordinates are particularly useful for circles. The equation of a circle of radius 5 with centre at the origin is simply

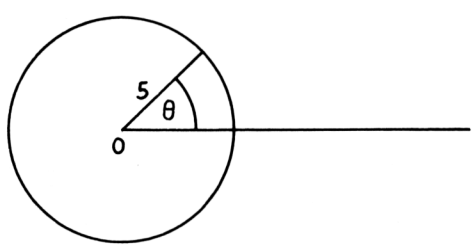


Figure A2.4 The polar equation of a circle radius 5

STRUCTURED PROGRAMMING IN MINIMAL BASIC

If you want to use this book with a microcomputer which does not have all the features of BBC BASIC, you will need to replace the missing structured features with simulated structures created from other BASIC commands.

REPEAT ... UNTIL ...

To produce this, use a `FOR ... NEXT ...` loop as in Program A3.1

```
10 REM simulated REPEAT loop
20
30 REM REPEAT
40 FOR I = 0 TO 1
50   )
60   ) loop action
70   IF NOT  repeat condition  THEN  I =  0 :  REM Loop
     counter is  re-set to prevent exit from loop
80   NEXT I
90 REM END REPEAT
```

Program A3.1

Example

Structured

```
10 REM wait 5 seconds
20
30 TIME = 0
40 REPEAT
50   UNTIL TIME = 500
```

Program A3.2

Simulated

```
10 REM wait 5 seconds
20
30 TIME = 0
40 FOR I = 0 TO 1
50   IF TIME < 500 THEN
       I = 0
60   NEXT I
```

Program A3.3

WHILE

This can be simulated using the same principle as the `REPEAT` loop, i.e. with a `FOR ... NEXT` loop with the counter re-set if the condition is not met.

```
10 REM simulated WHILE loop
20
30 REM WHILE
40 FOR I = 0 TO 1
     )
     )   loop action
50   IF NOT  while condition  THEN 90
```

```
      )
      )   loop action
80    I = 0
90    NEXT I
```

(You might prefer to make line 50: IF (condition) THEN GOSUB (action)).

Example

Structured

```
10 REM Look for a space
20
30 J = 1
40 WHILE MID$(T$, J, 1 ) <> " "
50   J = J + 1
60   ENDWHILE
```

Program A3.5

Simulated

```
10 REM Look for a space
20
30 J = 1
40 REM WHILE
50 FOR I = 0 TO 1
60   IF MID$(T$, J, 1)
       = " " THEN 90
70   J = J + 1
80   I = 0
90   NEXT I
```

Program A3.6

IF ... THEN ... ELSE ...

Two different simulations of IF ... THEN ... ELSE ... might be needed according to what form of IF ... THEN ... your BASIC supports.

If your IF ... THEN ... allows any valid BASIC expressions after THEN, the following technique will work:

IF condition THEN action1
IF NOT condition THEN else action

Example

Structured

IF expensive THEN buy ELSE notbuy

Simulated

IF expensive THEN buy
IF NOT expensive THEN notbuy

If you are only permitted to follow `THEN` with a line number, the simulation becomes unavoidably messy:

```
100 IF expensive THEN 110
110 REM 'IF'
120 notbuy : GOTO 150
130 REM 'ELSE'
140 buy
150 REM End ELSE
```

Program A3.7

Procedures

Provided that the procedure that you wish to simulate has no local variables and no parameters are passed to it, you can directly replace it with a subroutine.

If the procedure that you are replacing has local variables, use a subroutine and change the local variables to new variable names. Ensure that your program is not already using the new names.

If the procedure that you are replacing has parameters passed to it, replace it with a subroutine. Then, before each call to the subroutine, transfer the data to be passed to the variables used in the procedure.

Example

Structured

```
10 REM The something procedure

20
30 PROCsomething (7, "CAT")
40 END
50 DEF PROCsomething (N, L$)
60
70 procedure action
80 ENDPROC
```

Program A3.8

Simulated

```
10 REM The something
   subroutine
20
30 N = 7 : L$ = "CAT"
40 GOSUB 70
50 END
60
70 REM Subroutine
80
90 subroutine action
100 RETURN
```

Program A3.9

Procedures with local variables

These can be replaced with subroutines as described above but you must replace any local variables in the procedure with global variables in the subroutine. This may require a new variable in the subroutine. For example, here is Program 3.1 using procedures:

```
 10 REM Local variables
 20
 30 REM I in this loop is different
 40 REM variable from I in PROCdraw
 50 FOR I = 1 TO 10
 60   PROCdraw
 70   NEXT I
 80
 90 END
100
110 DEF PROCdraw
120 LOCAL I
130 FOR I = 1 TO 5
140   PRINT "*" ;
150   NEXT I
160 PRINT
170 ENDPROC
```

When we replace the procedure with a subroutine, the variable `I` in the procedure becomes global. Since `I` is already in use elsewhere in the program, we have to choose a new variable to replace `I` in the subroutine:

```
 10 REM Removing local variables
 20
 50 FOR I = 1 TO 10
 60   GOSUB 110 : REM Draw subroutine
 70   NEXT I
 80
 90 END
100
110 REM Draw subroutine
130 FOR J = 1 TO 5
140   PRINT "*" ;
150   NEXT J
160 PRINT
170 RETURN
```

£7.25

Structured Basic

BBC Basic offers superb facilities for writing well-structured programs. This book shows you how to use it.

The book assumes that you can write simple programs in *Basic* using the restricted *Basic* available on most micros. It also assumes that you have met all the problems and limitations of a restricted *Basic*: programs which are hard to control; complex and meaningless jumps using GOTO and GOSUB.

☐ The book is arranged in three parts. In the first part (Units 1 to 4) you are introduced to the special features available in *BBC Basic*. These enable you to write program modules whilst banishing GOTO.

☐ Then, in Unit 5, the ideas of the early units are used to show how to design a program step by step.

☐ The last part, Units 6 to 9, shows various applications of structured *Basic* to writing longer programs. These include data vetting systems, file handling, a data base and games.

☐ A cassette tape of all the main programs is also available.

Using the best that *BBC Basic* has to offer, programs can show their structure at a glance. They are easy to control, test and extend, and take far less time to prepare than in any other way. This book shows you how to do all that through making use of: * Top-down design * Modular design * Procedures * User defined functions * Local variables * TRUE and FALSE.